UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES

PAR
PATRICE GAGNON

VÉRIFICATION FORMELLE DE DIAGRAMMES UML : UNE APPROCHE BASÉE SUR LA LOGIQUE DE RÉÉCRITURE

AOÛT 2007

CE PROJET A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

M. Mourad Badri, Ph. D.
Directeur de Projet
Département de Mathématiques et d'Informatique
Université du Québec à Trois-Rivières

Mme Linda Badri, Ph. D.
Jurée
Département de Mathématiques et d'Informatique
Université du Québec à Trois-Rivières

M. Sylvain Delisle, Ph. D.
Juré
Département de Mathématiques et d'Informatique
Université du Québec à Trois-Rivières

VÉRIFICATION FORMELLE DE DIAGRAMMES UML : UNE APPROCHE BASÉE SUR LA LOGIQUE DE RÉÉCRITURE

Patrice Gagnon

SOMMAIRE

Le langage UML (langage semi-formel de modélisation graphique) est aujourd'hui le standard de l'industrie pour le développement des systèmes orientés objet. Il permet de modéliser et de décrire plusieurs vues complémentaires d'un même système. UML souffre, cependant, d'un manque de sémantique formelle. Les modèles UML développés pourront donc contenir des incohérences ou des inconsistances difficiles (voire impossibles dans le cas de certains systèmes complexes) à détecter manuellement.

Les méthodes formelles représentent une solution intéressante à ce problème. Les spécifications formelles auront pour effet d'éliminer les ambigüités au niveau de l'interprétation des modèles. La vérification formelle de modèles (model checking), quant à elle, tente de valider le comportement d'un système à l'aide de diverses propriétés énoncées à l'aide d'une logique (temporelle, etc.). La combinaison d'un langage de spécification formelle éprouvé et de techniques de vérification de modèles (*model checking*) solides permettra de valider formellement les modèles UML développés. Cette démarche permet d'éliminer certaines erreurs de façon précoce avant de passer aux phases de conception et de programmation. Elle peut également supporter la vérification du code ultérieurement.

Ce travail a consisté à développer un cadre formel permettant la translation de trois types de diagrammes UML vers une notation formelle du langage Maude. Cette description formelle est par la suite validée, avec plusieurs propriétés (logique temporelle linéaire LTL) du système à produire, à l'aide du vérificateur de modèles intégré de l'environnement Maude. Les trois types de diagrammes UML considérés sont les diagrammes de classes (structure), d'états-transitions (comportement individuel des objets) et de communication (comportement de groupe en termes d'interactions entre objets). L'environnement Maude est développé sur les bases de la logique de réécriture. Cette logique, possédant une forte sémantique mathéma-

tique, est conçue expressément pour modéliser les systèmes concurrentiels. L'approche développée a été évaluée sur trois études de cas concrètes : un système traditionnel, un système montrant de la concurrence interne, ainsi qu'un système avec plusieurs objets concurrentiels. Ce travail de recherche a fait l'objet de deux publications importantes [65, 32]. Par ailleurs, deux autres publications (conférence et journal) sont en cours d'évaluation.

FORMAL VERIFICATION OF UML DIAGRAMS : A REWRITING LOGIC BASED APPROACH

Patrice Gagnon

ABSTRACT

UML (semi formal and graphical modeling language) is nowadays the industry standard for object-oriented development. It offers the description of several views of a system, allowing the software engineer to model it from different angles. However, UML, suffers from a lack of formal semantics. This results in the fact that the developed models can contain inconsistencies and errors that can prove hard to find manually (or even impossible in the case of complex systems).

Formal methods offer an interesting solution to this problem. Formal specifications will have as an effect to eliminate ambiguities when reasoning on UML models. As for model checking, it attempts to validate the behavior of a system using different properties described with a given logic (temporal, etc.). The combination of a formal specification language and strong model checking techniques will allow the formal validation of the developed UML models. This approach allows the elimination of errors soon in the development process, well before the design and implementation phases. It can also support code verification afterwards.

This master's thesis consists on developing a formal framework allowing the translation of three types of UML diagrams into a formal description given in the Maude language. This formal description is then validated using several properties of linear temporal logic (LTL) about the system to be produced. The model checker used is the one included with the Maude environment. The three types of UML diagrams considered are class diagrams (static structure), state-transition diagrams (individual behavior of objects) and communication diagrams (collective behavior in terms of dynamic interactions between objects). The Maude environment is based on rewriting logic. This logic is very well adapted to model concurrent systems. The developed approach was evaluated with three concrete case studies : a traditional object-oriented system, a system showing elements of internal concurrency, as well as a

system showing several concurrent objects. This research was also the subject of two important publications [65, 32]. Furthermore, two more publications (conference and journal) are currently in evaluation.

REMERCIEMENTS

Je tiens à remercier M. Mourad Badri, professeur au Département de Mathématiques et d'Informatique de l'UQTR, d'avoir dans un premier temps réussi à me convaincre de me lancer dans un projet de Maîtrise, et ensuite d'avoir accepté de me superviser tout au long de mon travail. Ses précieux conseils et ses encouragements m'ont permis de réussir à compléter ce mémoire. Je veux également remercier Farid Mokhati, assistant professeur au Département d'Informatique de l'Université d'Oum-El-Bouaghi en Algérie, avec qui j'ai souvent collaboré lors de mes travaux. Je fais également une mention spéciale pour Mme Linda Badri et M. Sylvain Delisle, professeurs au Département de Mathématiques et d'Informatique de l'UQTR, d'avoir accepté de prendre un peu de leur temps pour lire et évaluer mon mémoire.

Je veux également remercier mon père, Jeannot, pour son éternel et inconditionnel soutien tout au long de mes études. Je mentionne aussi ma mère, Lise, qui veille sur moi même si elle n'est plus de ce monde. Je fais également une mention spéciale pour Mme Louise Roux et M. Jean-Jacques Lacroix. Leurs aide et suggestions auront grandement contribué à la rédaction de ce mémoire.

Bien qu'une Maîtrise soit un travail essentiellement personnel, discuter de notre travail avec autrui peut nous apporter de nombreuses bonnes idées. Parfois même, le seul fait d'en parler apporte une solution à un problème. À cet égard, je veux remercier sincèrement MM. Michel Charest, Daniel St-Yves, Yves Côté, Fadel Touré, Seybou Allagouma ainsi que Mme Josiane Lajoie. J'en oublie très certainement d'autres, mais considérez-vous également remerciés !

J'ai eu également de nombreux collègues au niveau du baccalauréat qui m'ont supporté tout au long de mes études de premier cycle. Pour n'en mentionner que quelques-uns, Mme Hanni Boulet ainsi que MM. Guillaume Giguère, Marc-André Dubé, Amine Bentchikou et Hassan Hilali ceux à qui je tiens à dire particulièrement merci. Et à tous les autres qui ont contribué plus ou moins directement à ce mémoire, encore une fois merci !

Table des matières

Liste des tableaux

Liste des figures

Liste des Abréviations et Sigles

Symbole	**Signification**
B	Méthodologie de spécification
BDD	Diagramme de Décision Binaire
BON	Méthodologie alternative à UML
CASE	Génie Logiciel assisté par Ordinateur
CTL	Logique Temporelle en arbre
CTL*	Union des logiques LTL et CTL
LaTeX	Langage pour la rédaction de documents
LTL	Logique Temporelle Linéaire
LTS	*Labelled Transition System*, Système de Transitions Nommées
MC	Vérification de modèles, *Model Checking*
MDA	Architecture basée sur les modèles, *Model Driven Architecture*
NIP	Numéro d'Identification Personnel
OCL	Langage de spécifications formelles, *Object Constraint Language*
OO	Orienté-Object
PROMELA	Langage d'entrée de SPIN
PTL	Voir LTL
SOO	Système Orienté-Object
SOOC	Système Orienté-Object Concurrentiel
SPIN	Nom d'un vérificateur de modèles
UML	Langage de modélisation orienté objet, *Unified Modelling Language*
Z	Méthodologie de spécifications

Introduction

Contexte

Depuis son invention au milieu du 20e siècle, l'informatique a connu une grande évolution, passant de courts programmes sur cartes perforées à des systèmes hautement automatisés, concurrentiels et complexes accomplissant une vaste gamme de tâches.

À toutes les époques, une grande question demeure toujours présente : comment s'assurer de l'exactitude d'un programme ? En effet, certaines des tâches accomplies par les systèmes informatiques sont parfois de nature critique, pour ne pas dire vitales. Pour ne citer que cela, prenons comme exemple un système de gestion d'un réacteur nucléaire. Une faille du système de sécurité informatisé du réacteur pourrait avoir des conséquences catastrophiques.

De nos jours, les systèmes développés sont de plus en plus complexes. Le paradigme de programmation orienté-objet a permis de nombreuses avancées et s'établit aujourd'hui comme le standard de l'industrie pour le développement logiciel. L'autre standard de l'industrie, UML (*Unified Modelling Language*), est un langage visuel (graphique) spécialement développé pour modéliser adéquatement un système orienté-objet (OO). Les diverses vues offertes par UML permettent de visualiser plusieurs aspects d'un même système. Ceci permet de mieux gérer la complexité du système.

Cependant, UML étant un langage à caractère plutôt visuel, il souffre d'un manque de sémantique formelle. En effet, les notations semi-formelles et visuelles d'UML peuvent entraîner des inconsistances au niveau des modèles développés.

En plus des systèmes OO classiques, très répandus, il existe également des systèmes

OO dits concurrentiels. Ces derniers ont la particularité d'avoir plusieurs processus à leur disposition. Chacun de ces processus accomplit une tâche quelconque.

Tous ces divers types de systèmes ayant leurs propres particularités, ils nécessitent donc un certain travail pour s'assurer du bon fonctionnement de leurs tâches critiques respectives. Certaines des erreurs qui peuvent se glisser au sein des modèles UML développés pour modéliser ces systèmes sont aisément détectables. Par contre, d'autres sont si subtiles qu'il est impossible de les détecter manuellement. Plusieurs techniques existent à ce niveau. Celles qui seront abordées dans ce mémoire sont les spécifications formelles et la vérification de modèles.

Les Spécifications formelles

Les spécifications formelles sont un ensemble de techniques qui proposent de représenter un système informatique à l'aide d'un langage dit formel. Un langage formel (ou langage de spécification formelle) est un langage hautement structuré basé sur une ou plusieurs théories mathématiques.

La Vérification de modèles

La vérification de modèles (ou *Model Checking* en anglais) est une technique automatique novatrice permettant de vérifier une propriété du comportement d'un système de façon automatique et exhaustive. Cette recherche exhaustive sur tous les chemins d'accès possible d'un système permet de vérifier si un, plusieurs ou aucun de ces derniers ne satisfont la propriété donnée. Les résultats fournis par le vérificateur permettront de retracer comment l'erreur, si erreur il y a, s'est produite et ainsi la corriger.

Approche proposée

Dans ce mémoire, une nouvelle approche est proposée pour la vérification de diagrammes UML. Tout d'abord, cette approche se propose de combiner les avantages des spécifications

formelles et de la vérification de modèles dans une seule et unique technique unifiée.

Cette technique est basée sur un processus de translation de diagrammes UML vers une notation formelle. La nouveauté qu'apporte la technique est de considérer à la fois les aspects statiques et les aspects dynamiques d'un système OO (de façon intégrée). Les aspects statiques d'un système seront représentés à l'aide d'un diagramme de classes UML. Quant à eux, les aspects dynamiques sont de deux natures : le comportement individuel des objets, ainsi que leur comportement collectif (interactions dynamiques).

Le comportement individuel réfère au comportement qu'adopte un objet à la réception de certains messages. Ce comportement est représenté à l'aide d'un ou plusieurs diagrammes d'états-transitions UML (habituellement, un diagramme par classe).

Le comportement collectif réfère quant à lui au comportement qu'adopte un groupe d'objets collaborant en vue d'accomplir une tâche qui leur est dévolue, en se partageant différentes responsabilités. Ce comportement collectif sera décrit à l'aide d'un diagramme de communication UML.

L'approche proposée introduit par la suite un processus incrémental pour la définition de propriétés du comportement individuel et collectif des objets d'un système OO. Ces propriétés sont ensuite vérifiées à l'aide d'un vérificateur de modèle.

Le langage formel retenu est le langage Maude. Ce dernier, basé sur la logique de réécriture, possède une forte sémantique mathématique. Il a spécifiquement été conçu pour prendre en considération les particularités des systèmes OO et des systèmes concurrentiels. Maude propose également de nombreux outils pour la vérification des systèmes. En plus d'être un langage formel, il est très versatile au niveau des simulations. Il propose également plusieurs outils au sein de son environnement, notamment un assistant à la preuve de théorèmes par induction (ITP) et un vérificateur de modèle.

Plan de travail

Ce mémoire est divisé en plusieurs parties. La première partie propose une revue de la littérature en deux volets. Le chapitre 1 présente de façon détaillée les spécifications formelles

et propose la revue de quelques techniques. Le chapitre 2 présente une revue approfondie de la technique de vérification de modèles, des principes sous-jacents et de diverses approches.

Le chapitre 3 propose une revue de l'environnement Maude et de sa logique sous-jacente : la logique de réécriture. Une revue des techniques de spécifications formelles et vérification de modèles avec cet environnement y est présentée.

L'approche proposée dans ce mémoire est décrite au chapitre 4. Les différentes études de cas réalisées, sur différents types de systèmes, sont présentées et discutées au chapitre 5. Ces études de cas, au nombre de trois, serviront à évaluer l'approche proposée. Par la suite, quelques remarques d'évaluation et quelques conclusions sont tirées.

Chapitre 1

Spécifications formelles

Tout un domaine d'étude en soi, les Spécifications formelles furent l'objet de nombreuses recherches. Ce chapitre tente de résumer les fondements de ces techniques, et d'exposer les techniques les plus pertinentes pour spécifier formellement un système logiciel.

1.1 Définition et concepts

Plusieurs méthodes de design et de documentation utilisent des techniques informelles pour atteindre leurs objectifs. Habituellement, seulement les diagrammes et le langage naturel sont utilisés pour décrire un système logiciel [7]. Comme le dit Bowen dans son livre [7], si une approche plus formelle est utilisée, on se retrouvera fort probablement avec une documentation plus complète et un design beaucoup plus simple.

La spécification formelle d'un système consiste en sa description à l'aide d'une notation à caractère mathématique. La principale raison derrière ceci est le fait que les notations mathématiques ont l'énorme avantage d'être précises alors que le langage naturel n'est que trop souvent ambigu et que les diagrammes le sont assez souvent aussi. Par contre, le fait que les notations utilisées, à caractère mathématique, soient si spécialisées entraine que peu de gens disposent de connaissances suffisantes pour les mettre en pratique efficacement [7, 19, 84]

Dans son livre, Jonathan Bowen [7] mentionne que l'avantage indéniable d'une notation formelle est qu'elle est précise et sans ambiguïté. Ainsi, une notation formelle fournira la des-

cription finale d'un système dans l'éventualité d'une mauvaise interprétation. Bowen définit les spécifications formelles comme étant la description d'un système à l'aide de notations mathématiques (confirmé par Clarke [19]).

Avec ces notations à caractère mathématique, on peut définir, à propos d'un système, diverses propriétés pouvant être liées, mais non limitées, au comportement fonctionnel, au comportement temporel, à des caractéristiques de performance et à la structure interne [19].

Les raisons qui pourraient inciter à spécifier formellement un système sont nombreuses. Tel que mentionné précédemment, le design sera plus simple, la documentation plus complète, et les ambiguïtés auront été éliminées [84, 7]. Il a clairement été démontré qu'utiliser un modèle formel d'un système en développement a souvent permis de détecter de sérieux problèmes au sein des notations informelles précédentes [84]. De plus, il sera plus facile d'explorer les différentes options de développement, plus rapidement et plus facilement, dans le processus de développement. Enfin, la possibilité de détecter des erreurs plutôt subtiles dès le début du processus de développement d'un logiciel, plus tôt que très tard, offre de nombreux avantages. Parmi celles-ci, on retrouve la réduction des coûts de développement et la satisfaction du client.

Généralement, les erreurs commises proviennent de trois différentes sources. La première est lorsque le programme n'est pas correct (il ne satisfait pas les spécifications du client). Le formalisme tend à prouver l'absence de telles erreurs. La seconde est lorsque le programme n'est pas adéquat (en quelque sorte une erreur dans les spécifications elles-mêmes). Le troisième type d'erreur est dû à la présence d'un bogue dans le système d'exploitation, dans le compilateur ou dans les périphériques. À ce moment, à moins que ces derniers ne soient spécifiés et vérifiés, ces erreurs ne peuvent être évitées. Par contre, il est très rare qu'il soit possible de spécifier et vérifier entièrement de larges systèmes. On se concentrera alors sur les modules, les parties ou les propriétés qui sont essentielles au système en développement.

Dans son document [84], Van Lamsweerde mentionne que les notations formelles sont souvent supportées par des outils automatisés pouvant être utilisés dans une variété d'objectifs. Parmi ceux-ci, les suivants sont certainement les plus importants :

– Génération de preuves à l'aide de techniques de preuve de théorèmes par déductions ;

- Vérifier si le modèle satisfait certaines propriétés, plus abstraites ou encore comportementales, ou produire un contre-exemple lorsque ce n'est pas le cas (techniques de vérification de modèles (voir le chapitre 2 pour plus de détails) ;
- Génération de séquences de tests ;
- Etc.

Il existe trois niveaux d'abstraction quand on parle de spécifications [7], soient :

Le niveau non formel : Le langage naturel que la majorité des gens utilisent. C'est le niveau le moins précis, qui fait apparaître le plus d'ambiguïtés ;

Le niveau semi-formel : On peut le caractériser par les langages de programmation actuels tels C++ ou Java. Bien que plus précis, ils laissent certains aspects ambigus ;

Le niveau formel : Il est le niveau d'abstraction le plus élevé, qui demande le plus grand effort d'apprentissage et de compréhension, mais qui ne laisse pratiquement rien d'imprécis dans ce qui sera spécifié.

Plusieurs éléments composent un langage de spécifications formelles. Ils sont composés, en premier lieu, d'un alphabet (une série de symboles reconnus du langage). La seconde composante essentielle est une grammaire, c'est-à-dire une série de règles qui permettront d'assembler les symboles en des phrases correctes. On appellera ceci le domaine syntaxique du langage. Le sens ou encore l'interprétation des formules est donné par le domaine sémantique. Ce domaine est composé d'une série de règles qui préciseront quel objet sert à remplir une certaine spécification donnée.

1.2 Approches

Les deux approches présentées ici sont associées à la méthodologie *RAISE*. *RAISE*, qui signifie *Rigourous Approach to Industrial Software Engineering*, est une méthode européenne qui, au bout de son développement, a donné naissance au langage RSL (*RAISE Specification Language*) et à une série d'outils [31, 70]. La méthode comprend une suite de techniques et de stratégies en vue du développement formel ainsi que des preuves. Funes et al. [31] ont

proposé une façon de spécifier formellement la structure statique d'un système en développement à l'aide du langage RSL. Leurs collègues, Meng et al. [70] se sont plutôt concentrés sur une façon de formaliser les diagrammes d'états-transitions. Bien que les deux approches soient distinctes, elles pourraient aisément être combinées et utilisées pour modéliser autant la structure statique du système que le comportement dynamique individuel des objets.

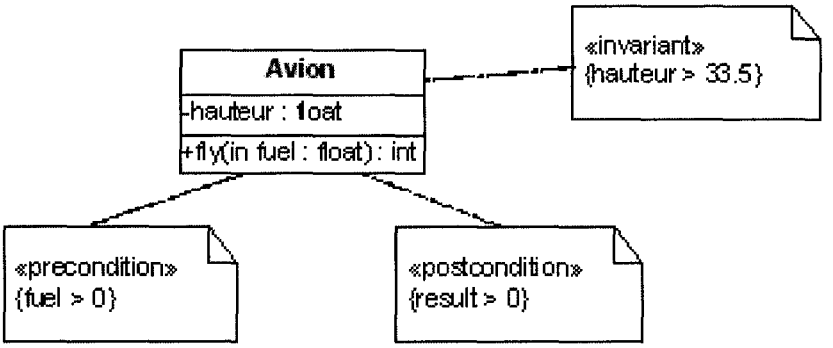


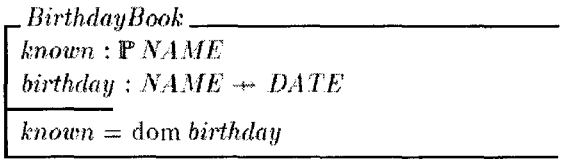
FIG. 1.1 – Exemple de notations OCL

OCL (*Object Constraint Language*) [86] est un langage de spécification formelle qui permet d'associer une ou plusieurs contraintes aux éléments d'un modèle orienté-objet. Étant un langage de type textuel, OCL se distingue par rapport à d'autres langages de spécifications (Z par exemple) qui sont basés sur la syntaxe symbolique des mathématiques. Particulièrement, OCL est formellement défini dans le document "*Object Constraint Language Specification*" [86] et fait partie intégrale de la spécification officielle d'UML. Dans le contexte de l'utilisation d'UML, une contrainte est une condition qui est imposée à un ou plusieurs éléments d'un modèle orienté objet. Il suffit qu'une seule condition soit invalidée pour confirmer que le modèle est incorrect ou inconsistant. Plus précisément, OCL est un langage formel qui permet la formulation de conditions, sous forme d'expressions booléennes, qui peuvent être vraies ou fausses. La spécification d'OCL n'impose pas de convention stricte sur l'endroit où on peut placer les contraintes OCL. Idéalement, les contraintes OCL (sous la forme d'invariants, préconditions et postconditions) devraient être fixées sur les modèles UML. Néanmoins, il arrive fréquemment qu'une contrainte soit présentée dans un autre document. Quand ceci est le cas, il est seulement important que le contexte (c'est-à-dire à quel élément UML la

contrainte s'applique) de la contrainte soit bien identifiée. La figure 1.1, tirée de [86], montre une classe UML nommée avion, pour laquelle on a défini les trois types de contraintes OCL : un invariant, une précondition et une postcondition.

Dans son document, Favre [30] propose de translater les diagrammes de classes et les paquetages UML vers une spécification formelle basée sur le langage NEREUS. Son approche est basée sur la méthodologie MDA (*Model Driven Architecture*). À la base, ce langage est centré vers les relations, c'est-à-dire qu'il traite les divers types de relations (Dépendance, Association, Agrégation, Composition) comme primitives de spécifications. La combinaison UML — OCL est utilisée pour maximiser les informations que peuvent contenir les modèles avant la translation vers NEREUS. Comme NEREUS est une sorte de plate-forme de communication entre UML et les langages orientés-objet, la méthodologie de Favre propose de générer automatiquement le code source du langage *Eiffel*.

Z et *Object-Z* sont probablement parmi les plus connues des approches proposées. Selon Bowen [7], bien que *Z* ne soit pas destiné aux systèmes orientés-objet, il existe de nombreuses approches permettant l'utilisation de cette méthode avec des systèmes orientés-objet. *Object-Z*, quant à lui, est expressément conçu pour de tels systèmes. Il est à noter par contre qu'*Object-Z* est une extension de *Z*. Les deux méthodes s'appuient sur des bases mathématiques, notamment sur la théorie des ensembles et des équations. Toujours selon [7], il existe de nombreux outils permettant d'utiliser *Z* et *Object-Z*. Tout d'abord, le premier outil intéressant à mentionner concernant ces deux langages connexes est très certainement LaTeX. En effet, les notations *Z* et *Object-Z* étant particulièrement illisibles, il est préférable de les afficher à l'aide de cet outil, qui les rend beaucoup plus compréhensibles [46]. La figure 1.2, tirée de [55], montre un très court exemple à cet effet.




```
\begin{schema}{BirthdayBook}
    known: \pset NAME \\
    birthday: NAME \pfun DATE
\ST
    known = \dom birthday
\end{schema}
```

FIG. 1.2 – Court exemple en *Object-Z*, avec LaTeXà gauche, et le programme original à droite

De nombreuses approches ont été proposées avec ces langages. Seulement du côté des systèmes orientés-objet spécifiés avec le langage *Object-Z*, les travaux suivants sont mentionnés : [46, 55, 43, 27, 81, 2]. En combinant les travaux de MacColl [55], de Dong [26] et de Joao [2], on peut alors parvenir à spécifier formellement la totalité d'un système objet, soit autant les aspects structurels que les aspects dynamiques individuels et collectifs (les travaux de Joao et al. [2] visent à formaliser les diagrammes de collaboration UML). Cependant, pris séparément, chacun de ces travaux ne considère qu'un seul aspect des systèmes orientés-objet, que ce soit structurel ou dynamique. *Object-Z* dispose également d'un vérificateur de type (*Type Checker*). Cet outil, expliqué en détail dans [43], s'appelle *Wizard* et permet la vérification de la conformité des types des paramètres ou valeurs de retour des fonctions des classes.

D'un autre côté, Paige et Brooke ont développé un outil de modélisation utilisant la méthodologie BON [74]. BON est un langage alternatif à UML pour le développement de systèmes orientés-objet. La particularité de BON réside dans le fait que cette méthodologie est principalement basée sur la classe comme élément fondamental au développement logiciel. Selon les auteurs, la particularité de leur outil automatisé, nommé *BON-CASE*, est le fait que celui-ci soit mis à la disposition de tous sur le web, par l'idéologie *Open source* où le code source de l'application est accessible à tout le monde. Ceci permet alors de rendre l'outil extensible, puisque chacun peut y apporter ça propre contribution.

L'outil *BON-CASE*, en plus de supporter le développement par la méthodologie BON, supporte l'utilisation des spécifications formelles par l'utilisation d'invariants de classes, de préconditions et de postconditions sur les méthodes des classes à l'aide du langage d'assertion de BON. À l'instar d'OCL, on dénote ici la présence d'un outil permettant de vérifier ces assertions [74]. La figure 1.3, tirée de [74], montre un exemple de classe décrite à l'aide du langage BON.

Dans un travail plus récent [75], les mêmes auteurs proposent une extension de leur outil par l'intégration de la méthodologie BON avec *Object-Z*. L'avantage, selon eux, est évident. Si, d'une quelconque façon, on pouvait rendre la génération de spécifications formelles *Object-Z* aussi facile que la génération de code, alors l'utilisation d'*Object-Z* en serait

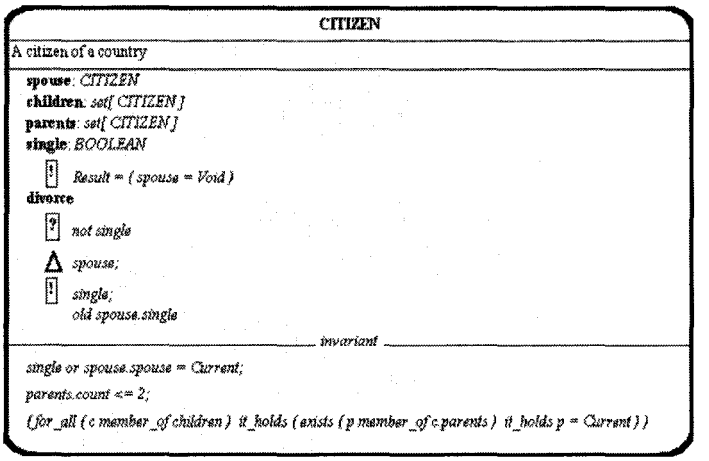


FIG. 1.3 – Exemple d'une classe décrite avec le langage BON

grandement facilitée. Puisqu'*Object-Z* offre de bien meilleurs outils automatiques pour la vérification que n'en offre BON, alors les possibilités sont d'autant plus intéressantes. Enfin, comme BON est une méthodologie de développement et qu'*Object-Z* n'est qu'un langage de spécification, la combinaison des deux approches a comme bénéfice d'apporter à ce dernier un processus de développement structuré. L'intégration de ces deux méthodes a été élaborée à l'aide de l'outil *BON-CASE* que les auteurs avaient développé auparavant.

La méthode B [11], développée par Abrial, est conçue pour la spécification, le design et le codage de systèmes logiciels. Tout comme *Object-Z*, B est basée sur des notations purement mathématiques. Ceci, une fois de plus, permet d'obtenir des spécifications du système en développement sans ambiguïté. La méthode B a aussi l'avantage d'être très répandue en Europe, et a fait l'objet de nombreux travaux. Elle est aujourd'hui supportée par plusieurs outils, notamment *Atelier B*.

La méthode B classique était basée sur les états. Avec le temps, Abrial ainsi que d'autres chercheurs ont développé une méthode B plus générale dans laquelle la notion d'événement prend plus d'ampleur. Les modèles basés sur les événements se sont avérés très utiles dans l'analyse des besoins, ainsi que pour la modélisation des systèmes distribués. Ainsi, une nouvelle méthode B basée sur les événements vit le jour. Z et B partagent de nombreuses racines communes, tout particulièrement en ce qui concerne les concepts mathématiques utilisés. La

figure 1.4, tirée de [11], montre un court exemple de notations B faisant référence à une classe *Button*.

MACHINE *Button*
SEES *Types*
USES *Lift*
VARIABLES
button, Button_State, Button_Floor, liftButton
INVARIANT
$button \subseteq BUTTON \wedge$
$Button\_State \in button \rightarrow BUTTON\_STATE \wedge$
$Button\_floor \in button \rightarrow FLOOR \wedge \ldots$
$liftButton \in lift \leftrightarrow button \wedge$
$\forall xx.(xx \in lift \Rightarrow$
$card(liftButton[\{xx\}]) \geq 2)) \wedge$
$liftButton^{-1} \in button \rightarrow lift \wedge$
$\forall(oo, fl).(oo \in lift \wedge fl \in FLOOR \Rightarrow$
$card(liftButton[\{oo\}] \cap Button\_floor^{-1}[\{fl\}]) = 2)$
DEFINITIONS
$FLOOR == (floor\_ground \ldots ground\_top)$
...
END

FIG. 1.4 – Un court exemple de notations B

Deux approches ont fait appel à la méthode B pour la formalisation de systèmes logiciels. Ces deux approches ont la particularité de modéliser à la fois les aspects statiques et les aspects dynamiques des systèmes orientés-objet.

La première de ces approches, proposée par Snook et al. [82], a été développée sous la forme d'un script pour l'outil CASE *Rational Rose*. *Rational Rose* est, pour sa part, un outil CASE très connu permettant la modélisation de systèmes à l'aide des artifices d'UML. Ce logiciel offre également la possibilité de générer automatiquement du code directement à partir des modèles développés. Dans cette optique, les auteurs de [82] ont quant à eux mis au point un script pour *Rational Rose*, développé pour générer automatiquement des spécifications formelles B directement à partir des modèles UML. Ils ont nommé leur outil *U2B*, faisant référence à la translation automatique de UML en des notations B.

Le second outil, *ArgoUML+B*, développé par Ledang et al. [53] comme une extension de l'outil *ArgoUML* développé par *Argo Tigris Corporation* [3]. Cet utilitaire est un outil CASE qui, tout comme *Rational Rose*, est conçu en vue d'utiliser les artifices d'UML. Leur outil

combine lui aussi la méthode de développement UML avec la méthode formelle B.

Il faut également mentionner les travaux de Laleau et Mammar [50, 49]. Ces deux auteurs proposent un kit d'outils développés pour l'environnement de modélisation *Rational Rose*. Les outils qu'ils ont développés proposent de dériver un programme SQL à partir d'un modèle UML. Cette translation se fait à l'aide d'une spécification B.

1.3 Discussion

Dans ce chapitre, nous avons vu tout d'abord ce que sont en général les spécifications formelles. Les concepts de base de ce type de méthode formelle y ont été introduits, ainsi qu'un bref historique. Par la suite, plusieurs techniques ont été exposées concernant tout particulièrement les systèmes orientés-objet.

La très grande majorité de ces approches ne considèrent cependant qu'un seul aspect des systèmes orientés-objet, que ce soit structurel ou dynamique. Par contre, la combinaison de certains travaux permet l'analyse complète d'un système. Ceci est vrai pour le langage *Object-Z* en particulier. En effet, en oeuvrant à partir de trois méthodes distinctes, une représentation fiable d'un système orienté-objet peut être obtenue. L'aspect structurel a été formalisé à l'aide de diagrammes de classes UML, tandis que les aspects dynamiques étaient aussi considérés à l'aide de diagrammes d'états-transitions UML pour le comportement individuel et de diagrammes de collaboration pour le comportement collectif des objets.

Des outils présentés, les deux derniers (respectivement *U2B* et *ArgoUML+B*) sont les seuls à considérer les aspects statiques et dynamiques à la fois. En effet, ces deux approches considèrent à la fois la structure statique des systèmes développés, par le biais de diagrammes de classes, tout en considérant le comportement individuel des objets à l'aide de diagrammes d'états-transitions.

L'apport de ce mémoire est de tenter le développement d'une approche permettant la translation de diagrammes UML vers une notation formelle dans laquelle seront considérés autant les aspects statiques que les aspects dynamiques (comportement individuel et collectif) d'un système en développement.

Chapitre 2

Vérification de Modèles

Tout comme le sont les Spécifications formelles, la Vérification de Modèles (*Model Checking*) est un domaine de recherche en soi. Ce chapitre tente d'introduire ce qu'est la vérification de modèles, autant classique que du point de vue des systèmes orientés-objet, et introduit les principales approches actuelles.

2.1 Définitions et Concepts

La vérification de modèles est une technique particulière des méthodes formelles ayant pour objectif la vérification automatique d'un modèle.

Sommairement, la Vérification de modèles consiste en la construction d'un modèle habituellement fini d'un système en développement, et de vérifier sur ce modèle si une série de propriétés y sont vérifiées [19]. En gros, la technique fait une recherche exhaustive au sein de l'espace des états du système [19, 24, 14]. D'une manière générale, deux descriptions distinctes d'un même système sont nécessaires pour effectuer la vérification [24], que l'on peut résumer par ce qui suit :

Le modèle : Le modèle consiste en une représentation du système en cours de développement, obtenu directement ou via une technique particulière (voir plus loin dans ce chapitre à ce sujet). Ce modèle doit être exécutable d'une certaine façon ;

Les spécifications : Les spécifications représentent l'expression des besoins envers un système, ou en quelque sorte ce que le système doit être en mesure d'accomplir lorsqu'il sera complété. Ces spécifications seront exprimées sous la forme de formules logiques temporelles et seront vérifiées au sein du modèle du système pour tenter de savoir si le système développé les satisfait ou non.

La figure 2.1 montre un bref aperçu de la technique à son origine, tel que présentée par Edmund M. Clarke dans l'introduction de son cours sur la vérification de modèles [20].

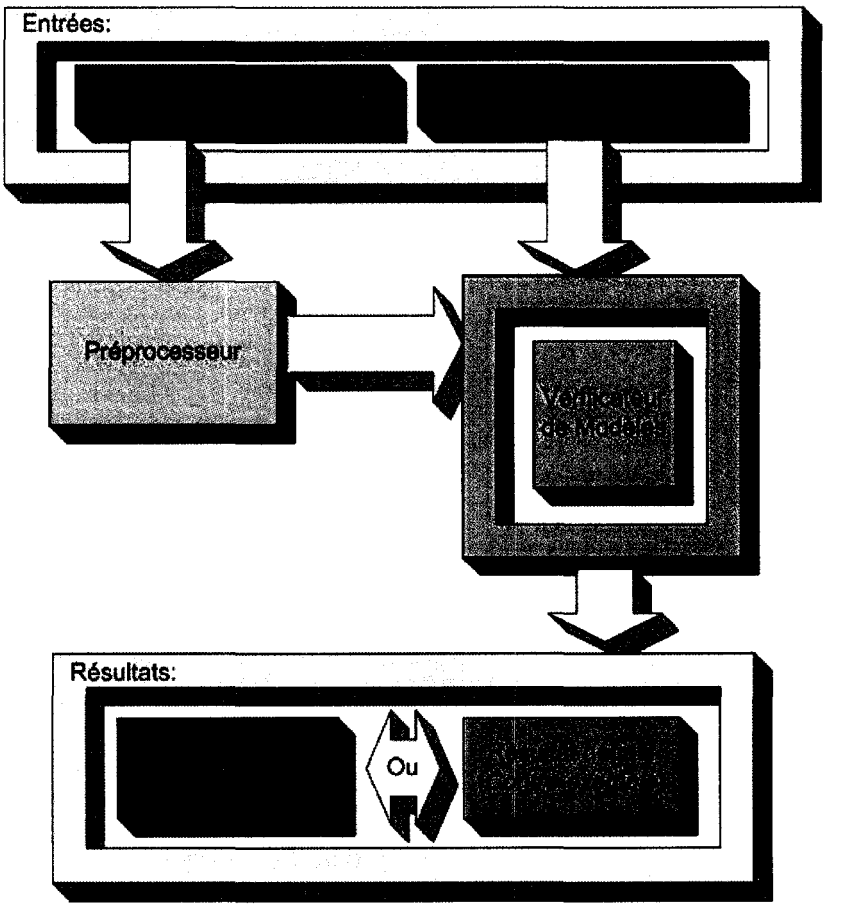


FIG. 2.1 – Aperçu du processus de vérification de modèles

Toujours selon Edmund M. Clarke [19], l'inventeur de la technique (voir la section suivante à ce sujet), la vérification de modèles offre de nombreux avantages :

- Aucune preuve écrite à faire (pour certains, l'avantage est indéniable) ;
- Rapidité (des résultats peuvent être fournis en quelques minutes seulement) ;
- Lorsqu'une erreur est rencontrée, le vérificateur fournit un contre-exemple montrant comment il a atteint l'état d'erreur ;
- Un système spécifié que partiellement ne cause pas de problème ;
- Les diverses notations logiques utilisées pour définir des propriétés ont la possibilité d'exprimer assez aisément les concepts de concurrence.

Bien sûr, les avantages d'une telle technique ne se limitent pas exclusivement à ces derniers. Les auteurs de [24] parlent d'une des techniques les plus révolutionnaires des dernières années au sujet de l'assurance qualité des logiciels. Par contre, il ne remplace pas la phase de test traditionnelle déjà bien établie dans le processus de développement. Les validations traditionnelles à l'aide de tests demeurent une étape essentielle du développement d'un logiciel d'envergure. Les deux techniques sont en fait complémentaires.

Il va sans dire que la détection des erreurs est un enjeu majeur dans le processus de développement d'un logiciel. En effet, des erreurs peuvent se glisser à tout moment au cours du processus de développement, que ce soit en phase de développement, aussi bien qu'en phase de codage. Il a aussi été clairement démontré que la correction de ces erreurs coûte très cher, et ces coûts augmentent au fur et à mesure que l'on avance dans le processus de développement.

Ainsi, plus les erreurs sont détectées tôt dans le processus de développement, plus les coûts associés à leur correction seront moindres. Donc, l'objectif est alors de détecter dès le début le plus d'erreurs possible afin de les éliminer du design du système. De plus, certaines erreurs sont si subtiles qu'elles sont pratiquement impossibles à détecter sans l'assistance d'un outil [60].

Pour toutes ces raisons, on aura tendance à privilégier l'implantation des techniques de vérification de modèles tôt dans le processus de développement. La figure 2.2 montre à quel moment il serait opportun d'intégrer la vérification de modèles au sein du processus de développement orienté-objet (tirée de [54]). Utiliser les techniques de vérification à ce moment au

cours du processus de développement aurait comme avantage de détecter tôt dans le processus les erreurs qui auraient pu se glisser dans les modèles développés, réduisant du fait même les coûts liés à la maintenance du système en éliminant les coûts de correction des erreurs et améliore grandement sa qualité.

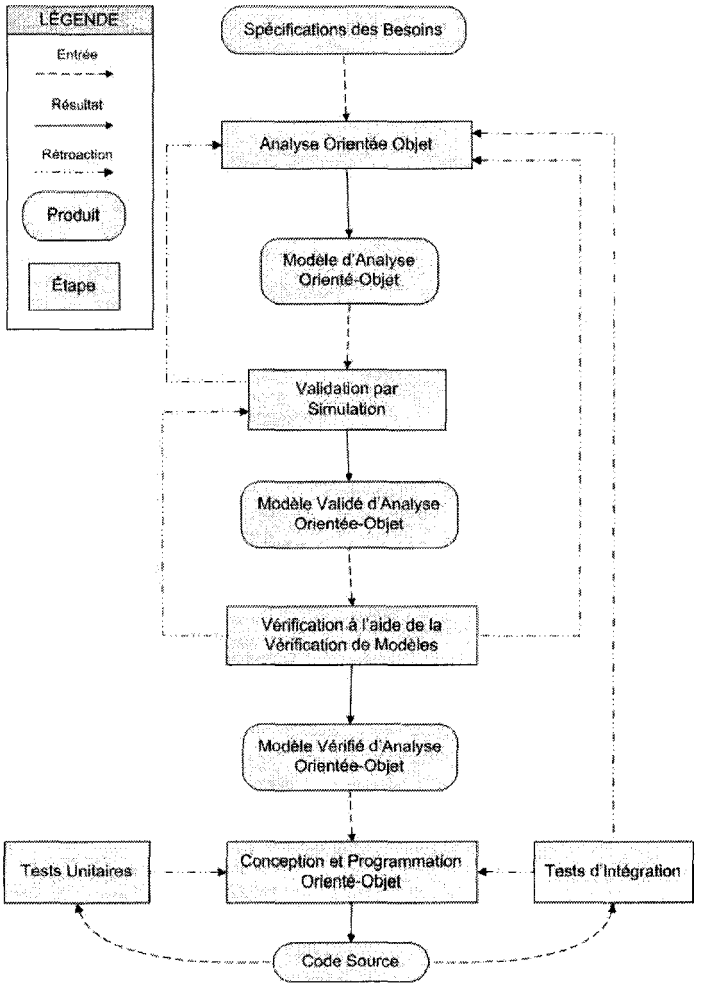


FIG. 2.2 – Intégration de la vérification de modèles dans le processus de développement orienté-objet

Originellement, la vérification de modèles, développée en parallèle par Clarke et Emerson en 1981 et par Quielle et Sifakis en 1982 [19, 14], était principalement utilisée pour la vérification de composants matériels [67, 19, 60]. De nos jours, la technique a été adaptée à

la vérification de logiciels, et tout particulièrement des systèmes orientés-objet et concurrentiels.

De façon un peu plus formelle, Gallardo et al. [24] définissent la tâche de vérification comme une tentative de vérifier si tous, quelques-uns ou aucun des chemins d'exécution du design courant du système en développement satisfont une propriété quelconque, que l'on veut désirable ou non désirable. Cette tâche est effectuée automatiquement en générant et analysant tous les états potentiels du système (dans la mesure où le nombre d'états est fini). Lorsqu'une propriété est vérifiée ou non (dépendant du cas dans lequel on se situe), l'outil retourne un résultat positif ou un contre-exemple illustrant le chemin qui a mené à cet état d'erreur.

Selon Wahl [85], si on désire vérifier des propriétés du type "Existe-t-il des chemins vers des états d'erreur ?", ceci est suffisant. Supposons que l'on veuille tester des propriétés comme les suivantes :

- L'instruction s est-elle exécutée infiniment souvent ?
- Est-ce que chaque Demande est suivie d'une Réponse ?
- Existe-il un point à partir duquel $x < 0$ n'est jamais vrai ?

Pour vérifier ce type de propriétés, la première définition de Wahl n'est plus suffisante. Il faut lui ajouter certains éléments.

Les premiers vérificateurs construisaient systématiquement le graphe des états du système tout au début, pour ensuite vérifier une certaine propriété, exprimée en logique temporelle, au sein de ces états.

Les vérificateurs modernes, quant à eux, ont tendance à transformer la propriété en automate afin de reconnaître les chemins d'exécution corrects et ceux qui sont incorrects. La vérification est faite au fur et à mesure que les états sont générés et stockés. C'est ce que l'on appelle les techniques *on-the-fly*. Bien sûr, l'espace des états doit toujours être fini. D'ailleurs, l'un des plus grands enjeux de la vérification de modèles est le problème de l'explosion de cet espace (notamment dans les systèmes industriels, qui ont tendance à être volumineux comparativement aux applications académiques étudiées). Plusieurs techniques, que l'on verra plus

loin, tentent d'apporter une réponse à ce problème. Voir la section 2.3 à ce sujet.

2.2 Éléments importants

Depuis le début de ce chapitre, on tente de définir ce qu'est la vérification de modèles. Au fur et à mesure de sa description, plusieurs éléments importants ont été abordés, tels que la logique temporelle, les propriétés, etc. Dans cette section, ces divers éléments seront décrits plus en détail. Il sera question des éléments suivants : les divers types de logique temporelle, un bref aperçu sur les propriétés et les divers types de vérification de modèles.

2.2.1 Types de logique temporelle

Plusieurs types de logique temporelle sont définis dans la littérature, chacune avec ses particularités, ses opérateurs et ses avantages. Seulement les plus reconnues sont présentées ici.

Logique temporelle propositionnelle

La logique temporelle propositionnelle (*Propositional Temporal Logic, PTL*), souvent nommée simplement Logique temporelle linéaire (*Linear Temporal Logic, LTL*) est utilisée dans la vérification de propriétés sur un seul chemin d'exécution.

Sans aller plus dans le détail pour le moment, les opérateurs X (*Next*) et U (*Untill*) sont les deux opérateurs de base de ce type de logique temporelle, et servent à définir une série d'autres opérateurs. Pour plus de détails à ce sujet, consulter le chapitre 3.

L'utilisation de la logique LTL est simple : elle est utilisée pour la vérification de propriétés concernant un seul comportement. Dans ce sens, la LTL n'exprime que des propriétés d'exactitude d'un système. Des propriétés dites de possibilité (une séquence du système satisfaisant une certaine propriété existe) relèvent plutôt de la logique en arbre.

Logique en arbre

La logique en arbre est principalement utilisée pour déterminer l'existence d'un chemin d'exécution du système qui satisfait une certaine propriété. Cette logique, appelée *Computation Tree Logic* en anglais, et simplement référée à CTL. Ce type de logique prend en considération les différents chemins dans le temps d'un système, et peut donc valider des propriétés globales du système.

On remarque alors que la grande différence entre les deux types de logique présentés, CTL offre des quantificateurs universels, E et G, respectivement "Il existe au moins un chemin..." et "Pour tous les chemins...". Ceci permet alors de quantifier les chemins d'exécution présentant les propriétés souhaitées.

Réunion des deux types de logique

Plusieurs autres types de logique ont été présentés dans la littérature. Plusieurs auteurs ont remarqué qu'on pouvait exprimer en LTL des situations impossibles à traiter en CTL. L'inverse est également vrai [60].

Alors, pourquoi ne pas joindre les deux types de logique en une seule ? Plusieurs tentatives ont été faites, mais les plus connues demeurent sans conteste CTL* et μ-Calculus [85]. Les deux consistent en la réunion des deux types de logique énumérés précédemment (grossièrement, on peut dire CTL* = LTL $\cup$ CTL. Le μ-Calculus est défini de façon similaire.). Ainsi, ces deux types de logique, différents quelque peu dans leur définition, offriront la possibilité d'exprimer des propriétés combinant à la fois des opérateurs de CTL et de LTL. Ces deux types de logique ne seront par contre pas présentés en détail.

2.2.2 Propriétés

Afin de mieux distinguer les concepts liés aux propriétés de logique temporelle, on introduira dans cette section plusieurs définitions qui apporteront de plus amples renseignements au sujet des propriétés de logique temporelle.

Les propriétés qui seront définies en vue de vérifier un système, qu'il soit logiciel ou

matériel, sont classées en catégories. Généralement, on reconnaît trois catégories principales, que Wahl résume ainsi [85] :

Sécurité : Les propriétés de sécurité visent principalement à s'assurer que le système respecte certaines conditions primordiales lors de son exécution. On pourrait, par exemple, vouloir s'assurer que l'exclusion mutuelle est toujours respectée au sein d'un système ;

Vitalité : Les propriétés de vitalité tentent de s'assurer du bon fonctionnement du système. On pourrait citer, dans le cas de systèmes concurrentiels, l'absence d'interblocage ou encore le fait que toute demande (*Request*) est suivie par une réponse (*Grant*), etc. ;

Égalité : Les propriétés d'égalité tentent de s'assurer que chaque tâche aura un temps d'accès au processeur équitable comparativement aux autres, notamment au sein des systèmes concurrentiels.

Enfin, les propriétés définies seront de deux natures distinctes, que l'on peut énoncer comme suit [85, 24, 16] :

Désirable : Une propriété dite désirable est une propriété que l'on souhaite que le système en développement soit en mesure de satisfaire. On parle également de propriété optimiste ;

Non désirable : Une propriété dite non désirable est une propriété que l'on ne désire pas retrouver au sein d'un système en développement. On parle également de propriété pessimiste.

2.3 Limitations et Solutions

Tel que vu précédemment, la Vérification de modèles est une puissante technique permettant d'augmenter la qualité des logiciels sur lesquels on l'applique. Cependant, la vérification de modèles, du moins dans son état actuel, souffre de deux problèmes importants.

En premier lieu, selon Wahl et Del Mar Gallardo et al. [85, 24], l'utilisation d'un vérificateur n'est pas si automatique que l'on pourrait le désirer, notamment à cause de l'obligation pour l'usager d'apprendre un nouveau langage (basé sur une certaine logique) en supplément à ses notations habituelles. Il existe de nombreux outils de vérification sur le marché.

Ceci entraîne le risque inhérent de créer des erreurs lors de la traduction d'un modèle dans ce langage. Pour le moment, ce problème n'est pas encore résolu, mais on tente d'y remédier notamment en créant des outils de translation automatisés vers un langage reconnu d'un vérificateur donné.

Le second problème de la vérification de modèles, beaucoup plus important, est celui connu sous le nom d'explosion de l'espace des états. Il est décrit avec de plus amples détails à la section suivante, et quelques tentatives de solutions sont données par la suite.

2.3.1 Explosion de l'espace des états

Ce problème, communément appelé l'*Explosion de l'espace des états* (mieux connu sous le nom de *State Explosion Problem* en anglais) [24, 60, 40, 19, 51] s'explique par le fait que la majorité des algorithmes explorent en entier les graphes des états, et que plus le système est complexe, plus le nombre de chemins possibles est grand.

- La vérification de modèles matériels souffre du problème d'explosion du nombre d'états du fait que la structure du modèle grossit exponentiellement plus le nombre de composants parallèles augmente ;
- Le même problème survient au sein des systèmes logiciels, lorsque des variables d'états ayant un domaine de valeurs fini, alors l'espace des états augmente aussi exponentiellement plus le nombre de variables augmente.

À titre d'exemple (tiré de [85]), un système orienté-objet classique avec n variables booléennes aura 2^n états possibles. De même, un système concurrentiel ayant n composantes avec chacune k états locaux aura ainsi k^n états globaux. Il est ainsi aisé de constater que le nombre d'états peut augmenter très rapidement.

Ainsi, la vérification de modèles est limitée par l'espace mémoire disponible sur un ordinateur donné. Plusieurs pistes de solutions ont été explorées pour remédier à cette situation. Dans ce qui suit, un bref aperçu des solutions les plus intéressantes est donné.

2.3.2 Solutions

Aux premiers développements de la technique, seule une représentation directe et explicite du système était utilisée. Aucune autre façon de représenter le système n'avait encore été développée. Ceci limitait énormément la taille des systèmes que l'on était en mesure de vérifier à l'aide de la vérification formelle de modèles.

Plusieurs tentatives ont été introduites pour remédier au problème d'explosion de l'espace des états. Dans ce qui suit, seulement les solutions principales et les plus intéressantes sont présentées.

Vérification de modèles symbolique : En 1987, McMillan développa une nouvelle façon de représenter les états d'un système. Son système de notation, les Diagrammes de Décision Binaire (*Binary Decision Diagrams, BDDs*), est basé sur la représentation à l'aide de formules booléennes. Ainsi, un système de transition (structure de Kripke) est représenté au complet à l'aide de formules booléennes. Il s'avère que les formules de logique temporelle exprimées en CTL s'utilisent très facilement avec un BDD, donnant ainsi naissance à la technique de vérification de modèles symbolique [60, 85] ;

Réduction partielle des ordres : La réduction partielle des ordres constitue une autre approche au problème d'explosion de l'espace des états. Ce type de technique est très efficace dans le cas des systèmes concurrentiels pour lesquels la communication entre processus est minimale. L'idée de la technique consiste alors en la réduction au maximum des états au sein de l'automate représentant le système. On élimine alors des noeuds doublés dans l'automate ;

Abstraction : L'abstraction est une autre technique permettant de représenter un système en vue de la vérification de modèle. Ce type de technique a plusieurs objectifs [85] :

- Rendre un système à états infinis en un système à états finis ;
- Réduire la taille d'un système ;
- Transformer un système adéquat afin de le représenter dans le langage d'entrée d'un vérificateur de modèle donné.

Une abstraction peut être exacte ou inexacte. Une abstraction exacte a la particularité

de préserver le comportement du système. Quant à l'abstraction inexacte, elle consiste principalement en l'ajout de propriétés comportementales au système (enlever des propriétés comportementales est très rare). Bien souvent, un modèle d'abstraction nécessitera un raffinement au fur et à mesure de son utilisation. La figure 2.3 présente un bref aperçu visuel d'une technique d'abstraction [85].

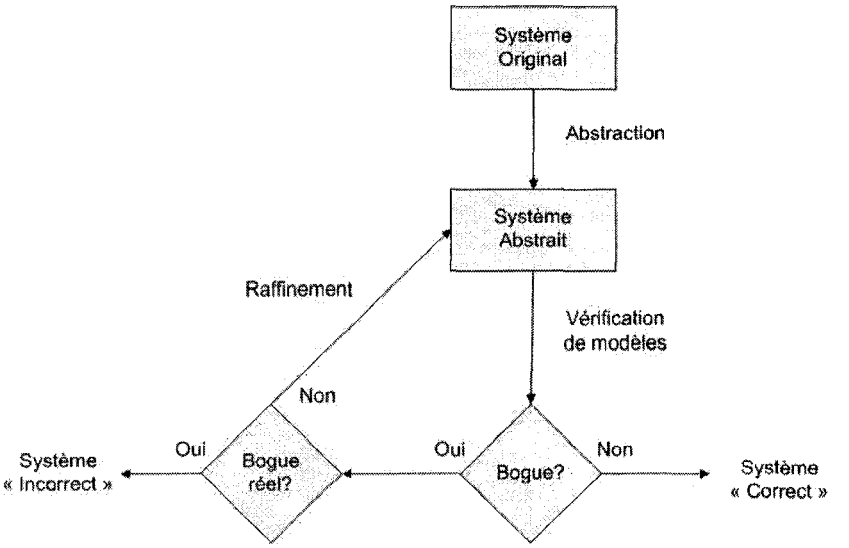


FIG. 2.3 – Aperçu visuel de la technique d'abstraction

Dans la figure 2.3, on remarque la présence d'une question sur la réalité d'un bogue : cela signifie que l'on cherche à savoir si le bogue que le vérificateur utilisé a détecté est réel (fait partie du système réel en développement), ou si ce bogue a été introduit en développant l'abstraction du système. Lorsqu'un bogue n'est pas réel, on utilise alors le raffinement du modèle abstrait pour le faire disparaître et relancer la vérification ;

Réduction symétrique : L'idée directrice de la réduction symétrique réside dans la permutation d'états et d'actions au sein d'un système. Une telle technique est valable lorsque la permutation de valeurs individuelles n'aura pas d'impact sur le comportement global du système. Par le biais de classes d'équivalences mathématiques, et pourvu que les propriétés de logique temporelle que l'on tente de vérifier soient elles aussi insensibles aux permutations, on peut parvenir à réduire considérablement la taille de la représen-

tation d'un système [60] ;

Vérification par composants : C'est une des façons les plus efficaces de contrer l'effet de l'explosion des états sur des systèmes à états infinis. On peut ainsi vérifier composant par composant un système, ou encore se limiter simplement aux composants les plus critiques [60, 48].

2.4 Outils et Approches

Dans les deux prochaines sections, quelques outils connus ainsi que les approches les plus intéressantes sont présentées.

2.4.1 Outils

Plusieurs travaux ont été faits dans le domaine de la vérification de modèles et plusieurs outils ont été développés pour supporter le processus de vérification qui peut devenir très lourd. Nous verrons, dans la section qui suit, quelques-uns de ces outils.

Vérificateurs traditionnels

SPIN : Sans conteste, SPIN est l'un des vérificateurs les plus connus et les plus répandus. SPIN, le vérificateur, prend en entrée des descriptions faites à l'aide du langage Promela, et possède les caractéristiques suivantes :

- SPIN se concentre explicitement sur la vérification de logiciels. Promela possède plusieurs constructions pertinentes à ce sujet, notamment le partage de mémoire pour la synchronisation entre processus.
- SPIN utilise les principes *on-the-fly* pour la construction de l'espace des états. Il ne considère alors que le sous-espace pertinent à la propriété étant testée ;
- Intégration complète de la logique LTL ;
- Support des techniques de preuves exhaustives et partielles ;

– SPIN utilise les ordres partiels et, rationnellement, des structures de stockage pour l'espace des états similaires aux BDD.

```
proctype MyProcessType()
{
  bit internalState = 0;
  do
    :: channelFromInitToMyProcess?0;
       internalState = 1;
    :: channelFromInitToMyProcess?1;
  od;
}
```

FIG. 2.4 – Exemple de code *Promela*

SPIN est un vérificateur de modèle qui, tel que le préconise l'approche, affiche un contre-exemple, ou une séquence d'étapes à suivre pour aboutir à une violation de propriété. Afin d'en faciliter l'utilisation, il existe une interface graphique, nommée Xspin, qui permet de créer et d'utiliser plus facilement les descriptions Promela (dont un bref exemple est donné à la figure 2.4) ;

SMV : Contrairement à SPIN, SMV fait l'objet de deux versions différentes. La première, Cadence SMV, s'est spécialisée dans l'utilisation de la logique LTL pour l'expression de propriétés sur les chemins d'exécution de logiciels, tandis que la seconde, CMU SMV, utilise le langage, CTL* vue précédemment. Pour l'implémentation des systèmes à espaces d'états finis, les deux versions utilisent les BDD et l'approche par vérification symbolique. La figure 2.5 montre un court bout de code SMV.

```
ordset TYPE 0..255;
module main() {
  x: INTEG ;
  init(x) :=0 ;
  next(x) := x+1 ;

  forall (i in TYPE)
    p[i]: assert F(x=i);

  forall(i in TYPE)
    using p[i] prove p[i-1];
}
```

FIG. 2.5 – Exemple de code pour le vérificateur SMV

SMV fait aussi l'implémentation de divers concepts utiles dans la vérification de mo-

dèles, notamment l'induction qui s'avère particulièrement utile dans les systèmes comportant plusieurs composantes ;

KRONOS : Sans toutefois s'y attarder beaucoup, KRONOS met principalement l'emphase sur les systèmes concurrentiels et les systèmes en temps réel. On utilise principalement une variété d'automates, les automates chronométrés, qui sont des automates standard étendus à l'aide de variables temporelles. Une logique particulière, la TCTL, similaire à la logique CTL mais pour laquelle on a pensé à l'inclusion de variables temporelles, a été développée. Principalement, KRONOS a les fonctionnalités suivantes :

- L'exécution du produit de plusieurs automates chronométrés ;
- Vérification d'une formule de TCTL sur un automate ;
- Réduction d'un automate en faisant abstraction du temps.

Des trois outils présentés jusqu'à maintenant, SPIN a le plus de potentiel et de possibilités. Ceci sera encore plus mis en évidence dans la section qui suit. D'un point de vue plus critique, nous pouvons en tirer ce qui suit. Des trois outils présentés jusqu'à maintenant, SPIN offre le plus de potentiel et de possibilités, tel qu'il sera démontré par l'étude des outils destinés à vérifier des diagrammes UML.

Outils pour UML

Trois autres outils suivront dans cette section. Leur particularité est majeure : ils tentent de faire le rapprochement entre les techniques traditionnelles de vérification de modèles et l'approche orientée objet, standard en programmation de nos jours. Les trois outils présentés sont, respectivement, vUML, UMLAUT et Bandera.

vUML : L'outil vUML devrait plus être considéré comme un traducteur. En effet, l'idée derrière ce petit outil est de traduire les diagrammes UML développés dans la phase d'analyse des besoins et de conception en langage *Promela* et ensuite de lancer l'analyseur SPIN sur ces nouvelles descriptions.

La particularité intéressante de vUML est la façon dont les résultats de contre-exemple sont retournés à l'utilisateur. En voici un exemple, sous la forme d'un diagramme de

```
Philosopher =
   UMLClass("Philosopher",
      attributes = None,
      behaviour  = StateMachine(
                     name       = "philosopherStateMachine",
                     top        = PhM,
                     states     = [thinking,eating,wfr,dropl],
                     transitions= [
                         Transition(
                            name    = "gLeft",
                            trigger = eAUTO,
                            source  = thinking,
                            target  = wfr,
                            effect  = Action(send=Send(left,eGet))
                         ),
                         Transition( [...] ),
                         Transition( [...] )
                         Transition( [...] )
                     ]
                  ),
```

FIG. 2.6 – Exemple de code *Promela* généré par vUML

séquence se terminant dans une impasse ;

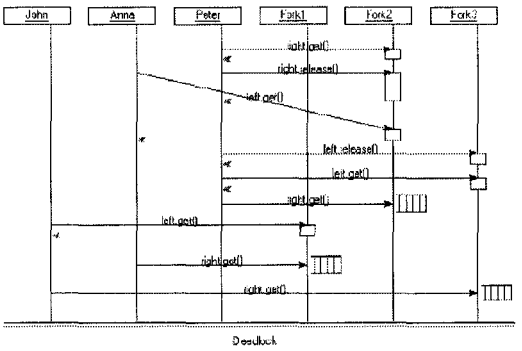


FIG. 2.7 – Contre-exemple fourni par vUML

UMLAUT : UMLAUT est un tout autre type d'outil, et n'est pas explicitement destiné à la vérification de modèles. En effet, l'objectif premier d'UMLAUT est la traduction ou plutôt la transformation de diagrammes UML pour atteindre divers objectifs. Ceux-ci sont autant de l'ordre de la phase d'implémentation que de la phase de vérification. Sans toutefois s'y attarder davantage, il est à noter que l'outil est encore en développement. De plus, le langage supporté qui sert à la vérification de modèles est, pour le moment, OCL. Enfin, l'outil possède une interface entièrement visuelle ;

Bandera : L'objectif primaire de Bandera est essentiellement de dériver un modèle orienté-objet à partir d'un programme Java. En effet, force est de constater qu'il est impossible de faire de la vérification directement en partant de code source Java. Le langage Java

possède une sémantique beaucoup trop vaste pour que ce soit fait à très grande échelle. Les mécanismes développés pour l'outil Bandera sont en réalité une série de mécanismes de réduction qui permettent de réduire du code source écrit en Java pour en arriver à une spécification dans un langage d'entrée d'un des divers outils de vérification de modèle disponibles actuellement.

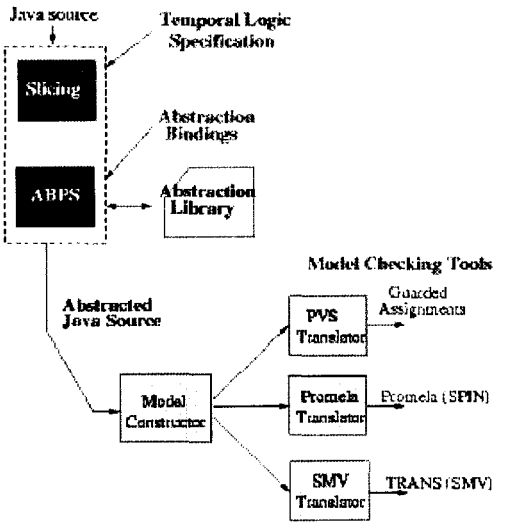


FIG. 2.8 – Approche supportée par l'outil Bandera

Les auteurs notent toutefois que Bandera n'a pu faire l'objet d'une analyse de leur part étant donné qu'il est encore au stade de développement. L'objectif des développeurs, comme le montre la figure 2.8, est la possibilité de réutiliser les vérificateurs SPIN et SMV vus précédemment.

Bien évidemment, plusieurs outils de vérification de modèles existent [85, 24, 83, 20]. Clarke, dans [20], mentionne notamment les deux premiers vérificateurs développés, EMC et Caesar. Il note aussi que SMV est le premier vérificateur à utiliser la vérification symbolique, et que SPIN utilise quant à lui la réduction partielle des ordres pour tenter de remédier au problème d'explosion de l'espace des états.

De nombreux autres outils existent. Le tableau 2.1, tiré de [85], en décrit quelques-uns en fonction de quelques caractéristiques importantes.

TAB. 2.1 – Quelques outils et leur portée respective d'origine

Outil	Représentation	Application	Logique
VIS	Symbolique	Matérielle	CTL
Cospan	Explicite	Varié	LTL
SPIN	Explicite	Logicielle	LTL
Murphi	Explicite	Matérielle	(Sécurité seulement)
NUSMV	Symbolique	Varié	CTL et LTL

2.4.2 Approches

Comme discuté précédemment [85, 19, 67], les premiers balbutiements de la vérification de modèles ont été élaborés dans les années 80. Les choses ont grandement évolué depuis, et les techniques utilisées sont très variées. Dans ce qui suit, les techniques les plus intéressantes sont présentées.

Publié voilà plusieurs années, l'article de Clarke et al. [18] fait un tour d'horizon somme toute complet des techniques de vérification disponibles à l'époque. Les auteurs y résument notamment les diverses logiques disponibles (LTL, CTL, etc.) et y introduisent le vérificateur SMV, encore disponible aujourd'hui. On y parle également de vérification symbolique et autres, tout en donnant un exemple concret d'application de SMV. La technique qu'ils proposent est en fait une évolution des techniques présentées dans [17] par quelques-uns des mêmes auteurs.

Dans [23], Das et al. présentent une approche très théorique pour la vérification de diagrammes d'états-transitions. Leur travail repose d'abord et avant tout sur la logique temporelle des arbres (CTL), pour laquelle ils prennent le temps d'introduire et d'expliquer tous les concepts nécessaires à la compréhension de leur approche. Ils présentent ensuite une série de routines permettant d'implémenter un système de vérification des diagrammes d'états.

Dans [15], les auteurs, Chechik et al, ont décidé d'étendre la logique temporelle LTL de façon à y inclure le raisonnement à partir de types de logique possédant plusieurs valeurs

de vérité. La figure 2.9 présente quelques exemples de ces types de logique. Ces différentes valeurs de vérité ne comportent pas seulement les valeurs *Vrai* ou *Faux* traditionnelles, mais aussi des valeurs comme *Possiblement Vrai*. Leur logique, χLTL, donnent les informations nécessaires pour raisonner à partir de telles logiques. Les auteurs ont d'ailleurs créé une extension au vérificateur MV-SMV pour compléter leur approche.

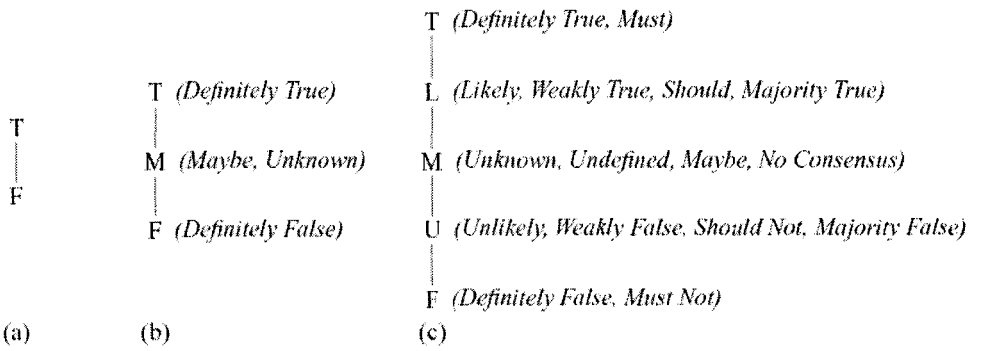


FIG. 2.9 – Exemples de types de logique à plusieurs valeurs de vérité : (a) traditionnelle, (b) à trois valeurs et (c) à cinq valeurs

Biere et al. [6] présentent les concepts nécessaires à la compréhension de la vérification de modèles limités. L'idée est simple : limiter à un certain nombre k les chemins possibles, et graduellement augmenter la valeur de k.

Dans leur article [34], Giannakopoulou et Magee utilisent le nouveau concept de *Fluent* pour introduire un nouveau type de vérification faite à partir des actions d'un système plutôt qu'à partir des états tel qu'il est plus coutumier de le faire. Un *fluent* peut être grossièrement défini comme un prédicat qui varie avec le temps, soit qu'il est activé par une action, ou terminé par une autre.

Jansamak et al. [42] présentent une nouvelle façon de représenter les diagrammes d'états à partir des *expressions régulières concurrentes* (CRE). On remarque aisément que leur approche est beaucoup plus théorique que certaines autres, et que de façon générale, leur technique s'applique plus facilement aux systèmes concurrentiels.

Dans [10], Canals et al. décrivent l'environnement de travail appelé NEPTUNE, qu'ils

ont développé pour supporter le processus de développement logiciel. La particularité de ce logiciel est qu'il contient un module, nommé *Checker*, qui vérifie le modèle UML développé contre certaines propriétés énoncées à l'aide du langage OCL. De façon analogue, dans [47], on propose d'utiliser les outils communs CASE (tel *Rational Rose*), de convertir ce modèle en format XML et d'utiliser cette représentation avec un outil que les auteurs ont construit pour vérifier la consistance du modèle développé.

Dans leur article [74], Paige et al. présentent l'outil d'aide au développement BON-CASE. Cet outil graphique se veut un support au développement, tels les autres outils CASE connus (par exemple, *Rational Rose*), tout en offrant la possibilité d'utiliser les méthodes formelles sur les modèles développés. Ceci est fait grâce au langage formel BON (qui est analogue à OCL et qui remplace également UML). L'outil permet alors de transformer le modèle développé en spécifications reconnues, données ici en JML et dont l'outil est pourvu d'un moteur de raisonnement. Les mêmes auteurs ont également présenté une intégration de leur approche avec la notation *Object-Z* dans [75].

Les auteurs des articles [5, 41] (Barnett et al. ainsi que Huizing et al.) proposent tous les deux des approches basées sur les invariants de classes. Les deux approches étant similaires, et [5] étant plus complète en soi, seule cette dernière sera étudiée de façon plus approfondie. De leur propre aveu, les auteurs on développé une méthode de raisonnement pour les invariants de classes, et leur méthode est principalement destinée à la vérification statique. Les auteurs élaborent donc une façon de procéder pour définir correctement des invariants sur les classes qu'ils développent. Ils incluent une façon de faire pour les situations courantes que l'on rencontre lors du développement, notamment l'héritage et les relations d'appartenance, etc. Leur façon de procéder consiste à ajouter à l'espace des états d'un système des variables auxiliaires qui permettent de déterminer lequel des invariants de classe est considéré fiable à ce moment. Le résultat est une hiérarchie d'appartenance de composants. Sur une autre note, les auteurs ne proposent pas de d'outil supportant leur méthodologie.

L'outil présenté par Shrotri et al. [80] utilise la logique temporelle des actions (TLA) et son langage associé de spécifications TLA^+. On y présente également l'outil que les auteurs utilisent pour faire leurs vérifications, le model checker TLC. De façon analogue, Kaveh et al.

présente le model checker LTSA dans [44] afin de détecter les interblocages éventuels dans les modèles définis. Les diagrammes d'états et les diagrammes de classes sont principalement utilisés. La logique et la sémantique sous-jacentes sont également présentées.

Dans [12, 13], Chan et al. font état de leur tentative de vérification sur un logiciel d'envergure (en l'occurrence, TCAS II, un logiciel d'avionique ayant pour objectif d'éviter les collisions aériennes). Leur approche est basée sur le langage de formalisation RSML, qui est en réalité une façon alternative à UML de représenter les diagrammes d'états. Ils utilisent ensuite, après conversion automatisée, le vérificateur de modèle SMV pour lancer leur analyse et leur vérification. Ils présentent également plusieurs options visant à réduire le nombre d'états d'un système, apportant ainsi des pistes de solutions au très connu problème d'explosion du nombre d'états dans la vérification symbolique.

Kholkar et al. [45] présentent un approche assez différente des autres. En effet, leur analyse se base sur différents diagrammes UML, tels les diagrammes de cas d'utilisation et les diagrammes de classes. Leur approche, principalement utilisée pour la détection d'inconsistances dans les modèles développés, est utilisée en conjonction avec l'environnement SAL. L'environnement SAL semble transformer les spécifications du système en langage reconnu par le vérificateur SMV qui, lui, détecte les erreurs automatiquement.

Merz et al. [78] présentent une approche qui propose d'utiliser l'environnement de travail HUGO, qu'ils ont développé, pour effectuer une série de vérifications sur des diagrammes d'états-transitions et des collaborations UML. La nouveauté de leur approche est la vérification des diagrammes d'états-transitions par rapport aux collaborations. Les diagrammes d'états-transitions doivent être écrits en Promela, le langage d'entrée du vérificateur SPIN. HUGO, quant à lui, interprète les entrées textuelles des collaborations, et fait ensuite appel à SPIN pour compléter les vérifications. D'emblée, les auteurs disent avoir de nombreux projets pour leur environnement. Ils veulent notamment être en mesure d'utiliser à meilleur escient les capacités de SPIN, d'utiliser les représentations XMI, qui sont de plus en plus répandues, et projettent également de pouvoir utiliser directement des spécifications OCL dans les diagrammes qu'ils utilisent.

Dans leurs articles [51, 52], Lam et al. formulent une approche complète basée sur le

π-Calculus. Le π-Calculus est en réalité la combinaison des logiques temporelles propositionnelle (LTL) et la logique temporelle des arbres (CTL). Leur approche est basée sur la formalisation des diagrammes d'états-transitions. Dans [51], les auteurs décrivent le procédé qu'ils utilisent pour décrire en π-Calculus les diagrammes d'états-transitions, ainsi que la façon d'implémenter le tout avec le vérificateur NuSMV. Ils présentent, à la fin, une étude de cas illustrant leur technique. Dans [52], ils complètent leur approche par le développement d'un environnement intégré permettant de supporter leur approche. Ils y décrivent le processus adopté pour développer les programmes d'interface entre les diagrammes d'états décrits avec le langage *Poseidon* et les systèmes qu'ils utilisent : MWB pour la vérification d'équivalences, et NuSMV comme environnement de vérification de modèles. Ils proposent également une étude de cas, ainsi que l'évaluation de chacun des deux environnements. Du point de vue de la vérification de modèles, leur choix s'est arrêté sur NuSMV, car cet environnement offre une solution relativement efficace au problème d'explosion de l'espace des états du système.

Dans leur article [24], Gallardo et al. développent une approche très intéressante. Sans toutefois proposer de solution pratique, ils élaborent une façon de procéder élégante pour effectuer des vérifications au niveau d'un système. Leur démarche se divise en trois étapes majeures : la vérification générale au niveau des diagrammes d'états-transitions du système, la vérification de comportements désirables du système et la vérification de comportements non désirables du système. Les auteurs proposent également de modéliser les comportements désirables et non désirables à l'aide de diagrammes de séquence. Selon les auteurs, cette façon de faire permet de s'assurer d'une très grande qualité du système, et comme leur approche préconise l'utilisation de diagrammes d'états-transitions, cette technique peut alors être mise en oeuvre très tôt dans le processus de développement et ainsi procurer les bénéfices énoncés dans la première partie.

Havelund et al. [38, 39, 37] ont développé un outil intéressant, nommé *Java PathFinder* ou plus simplement JPF. Cet outil est en fait un petit utilitaire qui permet d'appliquer la vérification de modèles directement à partir d'un code source Java. L'outil permet à un usager de définir diverses assertions au sein de son code source. L'outil convertit ensuite le

code source Java en langage Promela, le langage du vérificateur SPIN. Les assertions définies sont décrites comme étant des appels à des méthodes statiques d'une classe *Verify*. Les auteurs mentionnent également que leur outil permet de détecter les interblocages au sein des programmes Java.

2.5 Discussion

En résumé, la vérification de modèles se veut une technique automatique ayant comme objectif la vérification de propriétés de logique temporelle devant être satisfaites par le modèle d'un système en développement. De nombreuses approches à la vérification de modèles ont été développées depuis l'introduction de la technique au début des années 80. Quelques-unes de ces approches ont été présentées ici, en se limitant aux plus récentes et aux plus intéressantes.

Bien qu'importantes en regard de la vérification de modèles, les techniques matérielles sont plus ou moins intéressantes face à ce mémoire. Les plus intéressantes sont celles qui parlent de vérification logicielle. Parmi ces approches, celles qui sont les plus pertinentes à ce mémoire sont celles qui sont développées pour les systèmes orientés-objet. Le paradigme objet étant présentement le plus répandu, et UML le standard de l'industrie pour la modélisation logiciel, ce mémoire tente d'apporter une nouvelle technique pour la vérification formelle de ce type de système.

Les approches étudiées ont en commun le fait de ne considérer qu'un seul aspect des SOO. Parmi ces approches, les plus intéressantes sont celles proposées par Gallardo et al. [24] et Merz et al. [78]. Elles ont comme particularité de prendre expressément en considération les comportements individuels et collectifs des objets. Cependant, ces approches ignorent la structure statique des systèmes qu'ils étudient. Ce mémoire a pour objectif de développer une technique de vérification prenant en considération la structure statique et les aspects dynamiques d'un SOO. Les aspects dynamiques considérés portent à la fois sur le comportement individuel des objets et le comportement collectif des objets en termes d'interactions dynamiques.

Chapitre 3

Logique de Réécriture et Maude

Le système Maude est un environnement de programmation basé sur la logique de réécriture. La logique de réécriture, quant à elle, a été introduite dans le but de modéliser des systèmes informatiques, et plus particulièrement les systèmes concurrentiels. Ce chapitre introduira d'abord la logique de réécriture, pour ensuite aborder un peu plus l'environnement Maude, ses spécificités et son fonctionnement.

Maude étant basé sur de nombreux fondements mathématiques [62, 64, 22, 59], il est important de le présenter par une brève introduction à ces éléments. Dans les deux sections qui suivent, la logique des équations ensemblistes et la logique de réécriture feront l'objet d'une étude un peu plus détaillée. Par la suite, l'environnement Maude proprement dit sera discuté, ainsi qu'une brève introduction à la spécification formelle et à la vérification de modèles avec Maude [63, 64, 22, 20, 29, 73, 21].

3.1 Logique des équations ensemblistes

Selon Meseguer [61, 62], la logique des équations est particulièrement intéressante comme cadre mathématique pour les systèmes impératifs séquentiels, en plus d'être aisément exécutable.

La logique des équations ensemblistes est le type retenu par les créateurs de l'environnement Maude comme cadre formel mathématique pour la spécification des systèmes impératifs

séquentiels. La définition 3.1 introduit la signature d'une logique des équations ensemblistes.

Définition 3.1 (Signature de la logique des équations ensemblistes) *Une signature de Logique des équations ensemblistes est un triplet* $\Sigma = (K, \Sigma, S)$ *où :*

- K *est un ensemble de genres (en anglais,* kinds*) ;*
- Σ *est une famille de symboles de fonctions de* $K^* \times K$*-indexé avec* $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$ *où chaque* $f \in \Sigma_{k_1 \ldots k_n, k}$ *est noté par* $f : k_1 \ldots k_n \rightarrow k$ *;*
- $S = \{S_k\}$ *est une famille d'ensembles disjoints* K*-indexé, dont les éléments sont nommés sortes (en anglais,* sorts*).*

Ainsi, une formule Σ-atomique est soit une Σ-équation $t = t'$ ou un Σ-ensemble $t : S$, sachant que $t, t' \in T_\Sigma(X)_k$ et $s \in S_k$, où T_Σ est une théorie de logique pour Σ. La définition 3.2 introduit la notion de théorie d'équations ensemblistes.

Définition 3.2 (Théorie d'équations ensemblistes) *Une théorie d'équations ensemblites est une paire* (Σ, E) *avec* Σ *une signature de logique d'équations ensemblites et* E *un ensemble de* Σ*-équations et* Σ*-ensembles ayant la forme suivante :*

$$(\forall X) \quad t = t' \quad \Leftarrow \quad u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m$$
$$(\forall X) \quad t : s \quad \Leftarrow \quad u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m$$

Cela signifie que les phrases de E sont des *Clauses de Horn* universellement quantifiées dont les atomes sont des Σ-atomes. Une clause de Horn est une formule logique comprenant plusieurs propositions, mais dont une seule est positive. On prendra alors en considération les conventions de notations suivantes :

Sous-Types : La notation de sous-type (en Maude, *subsort*), signifie que pour les sortes $s, s' \in S_k$, la déclaration $s < s'$ veut en réalité dire $(\forall x : k) \quad x : s' \Leftarrow x : s$;

Genres : Les genres (en Maude, *kinds*) ne sont pas déclarés explicitement. Ils sont plutôt inférés par le système et sont identifiés à l'aide des classes d'équivalences $(S, \leq)$;

Opérations : Si $f \in \Sigma_{k_1,\ldots,k_n,k}$ et $s_1 \in S_{k_1}, \ldots, s_n \in S_{k_n}$ alors la déclaration $f : s_1 \ldots s_n \rightarrow s$ signifie $(\forall x_1 : k_1, \ldots, x_n : k_n) \quad f(x_1, \ldots, x_n) : s \Leftarrow x_1 : s_1 \wedge \ldots \wedge x_n : s_n$;

Variables : La notation $(\forall x : s, X) \quad a \Leftarrow a_1 \wedge \ldots \wedge a_n$ est une notation abrégée pour la Σ-phrase $(\forall x : k, X) \quad a \Leftarrow a_1 \wedge \ldots \wedge a_n \wedge x : s$ où $s \in S_k$, a et les a_j sont des Σ-atomes. Ainsi, la notation $f : s_1 \ldots s_n \rightarrow s$ est équivalente à $(\forall x_1 : s_1, \ldots, s_n) \quad f(x_1, \ldots, x_n) : s$.

L'objectif ultime est d'être en mesure de faire un raisonnement logique à partir de ce type de théorie. Pour ce faire, plusieurs règles d'inférence sont introduites afin d'aider la déduction. Ces règles sont énumérées ci-après.

1. **Réflexivité :** Pour l'ensemble E, on a $t = t$;
2. **Symétrie :** Pour l'ensemble E, on a $t = t'$ et $t' = t$;
3. **Transitivité :** Pour l'ensemble E, si $t = t'$ et $t' = t''$ sont vraies, alors $t = t''$ est vrai ;
4. **Congruence :** Pour l'ensemble E, sachant que $t_i = t'_i$ pour $1 < i < n$, alors $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$ est vraie en prenant pour hypothèse que $f : k_1 \ldots k_n \rightarrow k \in \Sigma$ et que les termes $t_i, t'_i \in T_\Sigma(X)_{k_i}$ pour $1 < i < n$;
5. **Appartenance :** Pour l'ensemble E, sachant que $t = t'$ et que $t : s$, alors $t' : s$;
6. **Modus Ponens :** Le Modus Ponens est une forme très connue de raisonnement logique. Il prend la forme suivante :

   Si A, alors B. A. Alors B.

Les éléments précédents définissent l'essentiel de la logique des équations ensemblistes. Ces notions sont suffisantes. Elles constituent les éléments essentiels à la compréhension de l'environnement Maude. Meseguer et son équipe [64, 22] ont utilisé ces notions pour la définition des modules fonctionnels de Maude, que nous verrons un peu plus loin.

3.2 Logique de Réécriture

Dans ce qui suit, une brève introduction à la logique de réécriture [61, 62] sera donnée afin de mieux saisir son fonctionnement lorsque viendra le temps de la mettre en application. Les éléments définis ici sont tirés d'un texte de Meseguer [63]. La définition 3.3 introduit le concept de théorie de réécriture.

Définition 3.3 (Théorie de Réécriture) *Une théorie de réécriture est un triple* $\mathcal{R} = (\Sigma, E, R)$ *où :*

- (Σ, E) *est une théorie d'équations ensemblistes ;*
- R *est un ensemble de règles de réécritures conditionnelles nommées ayant la forme suivante :*

$$r : t \longrightarrow t' \Leftarrow \left(\bigwedge_i u_i = u_i'\right) \wedge \left(\bigwedge_j v_j : s_j\right) \wedge \left(\bigwedge_l w_l \longrightarrow w_l'\right)$$

pour lequel r *est un identificateur pour la règle, et où il existe un ensemble fini* X *de variables pour lequel* $t, t' \in T_\Sigma(X)_k$ *pour un genre (*kind*)* k*, ainsi que tous les* Σ*-termes* $u_i, u_i', v_j, w_l, w_l'$ *dans la condition sont également dans l'ensemble* $T_\Sigma(X)_k$*. La condition est la conjonction d'équations, d'équations ensemblistes et de réécritures.*

Lorsqu'une règle de réécriture ne comporte aucune condition, cette règle est alors appelée *inconditionnelle.* Ainsi, on peut introduire la définition d'une théorie de réécriture générale. La définition 3.4 introduit cette notion :

Définition 3.4 (Théorie de réécriture générale) *Une théorie de réécriture générale est un quadruplet* $\mathcal{R} = (\Sigma, E, \phi, R)$ *où :*

- (Σ, E) *est une théorie d'équations ensemblites avec, disons, les genres* K*, les sortes* S *et les opérations* Σ *;*
- $\phi : \Sigma \longrightarrow \mathcal{P}_{fin}(\mathbb{N})$ *est une famille indexée de fonctions de* $K_* \times K$ *qui assigne à chaque* $f : k_1 \ldots k_n \longrightarrow k$ *dans* E *l'ensemble fini* $\phi(f) \subseteq \{1, \ldots, n\}$ *de positions d'arguments figés ;*
- R *est l'ensemble des règles conditionnelles nommées de réécritures (quantifiées universellement) ayant la forme (avec les paires* t, t' *et* w_l, w_l' *ayant le même genre) :*

$$r : t \longrightarrow t' \Leftarrow \left(\bigwedge_i u_i = u_i'\right) \wedge \left(\bigwedge_j v_j : s_j\right) \wedge \left(\bigwedge_l w_l \longrightarrow w_l'\right)$$

De façon similaire à une théorie d'équations ensemblistes, une théorie de réécriture possède une série de règles d'inférence. Elles sont la Réflexivité, l'Égalité, la Congruence, le Remplacement et la Transitivité. La figure 3.1 permet de visualiser chacune de ces règles.

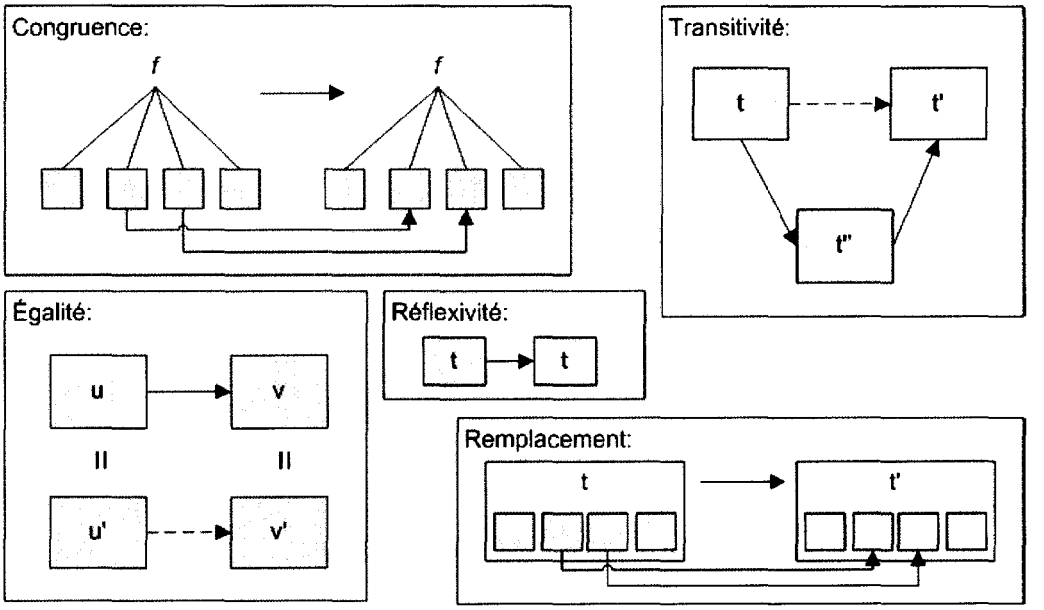


FIG. 3.1 – Visualisation des règles d'inférence d'une théorie de réécriture

Les quelques notions précédentes constituent les éléments les plus importants par rapport à la logique de réécriture. Les principales notions mathématiques pertinentes à la logique de réécriture ayant été discutées brièvement, l'environnement Maude peut être introduit.

3.3 Le système Maude

Maude est un environnement de programmation destiné à la modélisation de diverses notions relevant de plusieurs domaines [29, 63, 64, 22, 59, 73]. Ce langage, basé sur la logique de réécriture vue précédemment, a été développé en fonction de trois grands objectifs : la simplicité, l'expressivité et la performance.

Simplicité : Un programme devrait être le plus simple possible et aussi aisé à comprendre que possible. Il devrait également posséder une sémantique très claire ;

Expressivité : Il devrait être possible d'exprimer naturellement une vaste gamme d'applications. Il devrait être aussi possible de le faire autant pour un programme déterministe que pour un programme hautement concurrentiel ;

Performance : Il devrait être possible d'utiliser un langage comme spécification exécutable d'un modèle, mais également comme un véritable langage de programmation compétitif.

La plupart du temps, la simplicité et la performance sont présentes dans n'importe quel langage. Ils vont de pair, ils sont des alliés naturels. Par contre, maximiser l'expressivité du langage est sans aucun doute l'un des avantages les plus marquants du langage Maude. Certains diront certainement qu'un langage développé spécifiquement pour un domaine donné a l'avantage de simplifier les notations pour les initiés. Dans ce contexte, Maude devrait alors être vu comme un *métalangage* avec lequel il est très aisé de développer un langage spécifique.

Il est également important de noter que Maude permet de programmer à deux niveaux différents : *Core Maude* et *Full Maude*. On peut différencier les deux niveaux par ce qui suit :

Core Maude : Core Maude est le niveau de base de Maude, programmé directement en C++. Il implémente toutes les fonctionnalités de base du logiciel, les modules fonctionnels et les modules systèmes ;

Full Maude : Full Maude est le niveau supérieur. Programmé en Core Maude, Full Maude est en réalité une extension du premier niveau, avec lequel il est possible de combiner des modules pour perfectionner le développement.

Full Maude offre de nombreux avantages. Il est notamment utilisé avec le paradigme de programmation orienté-objet et les modules orientés-objet Maude. Par contre, dans le cadre de ce mémoire, la notation orientée-objet Maude sera utilisée sous sa forme de modules systèmes, et seul *Core Maude* sera utilisé.

3.3.1 Modules Maude

Le développement Maude se fait par l'écriture de divers modules décrivant le système que l'on écrit. Ainsi, l'unité de base pour le développement Maude est le module. Il en existe trois types : les modules *fonctionnels*, les modules *système* et les modules *orientés-objet*.

Module fonctionnel : Les modules fonctionnels sont principalement utilisés pour la définition de types de données et d'opérateurs via la théorie des équations ;

Module système : Les modules systèmes, quant à eux, sont utilisés pour définir des théories de réécriture. D'une certaine façon, tout comme la logique de réécriture est plus générale que la logique des équations, un module système englobe un module fonctionnel ;

Module orienté-objet : Ce dernier type de module est développé spécifiquement pour le paradigme de développement orienté-objet qui est aujourd'hui la norme sur le marché. Ce type de module est utilisé au niveau de Full Maude. Par contre, il est à noter que les modules orientés-objet peuvent être réduits en des modules systèmes. C'est ce procédé qui sera utilisé tout au long de ce mémoire.

3.3.2 Syntaxe de Maude

Dans ce qui suit, quelques brefs exemples de code Maude seront donnés afin d'illustrer le fonctionnement du langage. Ces exemples permettront de mieux saisir le fonctionnement des programmes qui seront étudiés ultérieurement dans le cadre du développement de la méthode que propose ce mémoire.

Rappel : Logique de réécriture

Bien que la logique de réécriture ait été étudiée plus en profondeur dans une section précédente, elle est réintroduite ici avec une notation beaucoup plus simple, afin d'en clarifier son utilisation.

La logique de réécriture, dotée d'une sémantique saine et complète, a été introduite par Meseguer [61]. Elle permet de décrire les systèmes concurrents [63, 59, 29, 64, 22]. Cette logique unifie tous les modèles formels qui expriment la concurrence [61]. De plus, d'un point de vue abstrait et mathématique, la logique de réécriture peut être vue comme un graphe orienté [59]. Dans la logique de réécriture, les formules logiques sont appelées règles de réécriture. Elles sont de la forme suivante :

$$R : [t] \rightarrow [t'] \quad \text{if} \quad C$$

La règle R indique que le terme t devient (se transforme en) t' si une certaine condition C est vérifiée. Le terme t représente un état partiel d'un état global S du système décrit. La modification de l'état global S du système vers un autre état S' est réalisée par la réécriture en parallèle d'un ou plusieurs termes qui expriment les états partiels. L'état distribué d'un système concurrent est représenté comme un terme dont les sous-termes représentent les différents composants de l'état concurrent. La structure de l'état concurrent peut avoir une variété de représentations équivalentes d'un terme parce qu'elle satisfait certaines lois structurelles (en mathématiques, la classe d'équivalence). Par exemple, dans un système orienté-objet, l'état concurrent, qui est usuellement appelé configuration, a la structure d'un multi-ensemble d'objets et de messages.

Court programme en Maude

La figure 3.2 présente un court programme écrit à l'aide du langage Maude.

```
1.  sort Configuration .
2.  sort Object .
3.  sort Msg .
4.  subsort Object < Configuration .
5.  subsort Msg < Configuration .
6.  op null : -> Configuration .
7.  op _ _ : Configuration Configuration -> Configuration
             [assoc comm id : null] .
```

FIG. 3.2 – Exemple d'une portion de programme Maude

La portion du programme illustrée par la figure 3.2 donne la définition de trois types : *Configuration*, *Object* et *Msg* (lignes 1 à 3). Dans les lignes 4 et 5, *Object* et *Msg* sont définis comme des sous-types (sous-ensemble) de *Configuration*. Les objets et les messages sont, en fait, des multi-ensembles singletons de configurations. Des configurations plus complexes sont générées à partir de l'application de l'union sur ces multi-ensembles singletons (objets

et messages). Dans le cas où il n'y a ni messages flottants, ni objets en vie, le système se trouve alors dans le cas d'une configuration vide (ligne 6, l'opérateur *null*). La construction d'une nouvelle configuration, en termes d'autres configurations, se fait à l'aide de l'opération de la ligne 7. Nous remarquons que cette opération n'a pas de nom et les deux barres de soulignement indiquent les positions des deux paramètres de type *Configuration*. Cette opération qui est, en fait, l'union multi-ensemble, satisfait les lois structurelles d'associativité et de commutativité (tel qu'indiqué par les définitions entre crochets). Elle possède, par ailleurs, un élément neutre *null*. À titre explicatif, si nous avons un message *M1* qui représente une configuration et un objet < *O* : *C* | *atts* > (noter que *O* est un identifiant d'objet, *C* est sa classe d'appartenance et *atts* est la liste de ses attributs) qui représente lui même une autre configuration, alors il est possible de construire une troisième configuration en termes de ces deux configurations : *M1* < *O* : *C* | *atts* >, celle-ci étant équivalente à la configuration : < *O* : *C* | *atts* > *M1* parce que l'opération _ _ est commutative.

Modules fonctionnels

Tel que discuté précédemment, il existe trois types de modules en Maude : les modules fonctionnels, systèmes et orientés-objet. Dans ce qui suit, un très court programme Maude sera défini à l'aide d'un module fonctionnel. La figure 3.3 montre ce court programme.

```
fmod PEANO-NAT is
  sort Natural .
  op 0 : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op _ + _ : Natural Natural -> Natural .
  vars N M : Natural .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
endfm
```

FIG. 3.3 – Le module fonctionnel *PEANO-NAT*

Ce court programme montre en réalité une façon alternative pour définir les nombres naturels. On le fait ici à l'aide de la fonction Successeur. Ainsi, un seul nombre de base existe,

0, et les autres sont définis à l'aide de la fonction successeur (l'opérateur s). Deux équations sont également définies : un nombre naturel additionné à 0 donne ce nombre naturel, et un nombre naturel additionné au successeur d'un autre nombre naturel donne le successeur de l'addition des deux nombres naturels.

Modules systèmes

Le second type de module intéressant est le module système. Par rapport au module fonctionnel, ce dernier incorporera des règles de réécriture. Tout comme la logique de réécriture qui inclut la logique des équations ensemblistes, les modules systèmes incluent les modules fonctionnels.

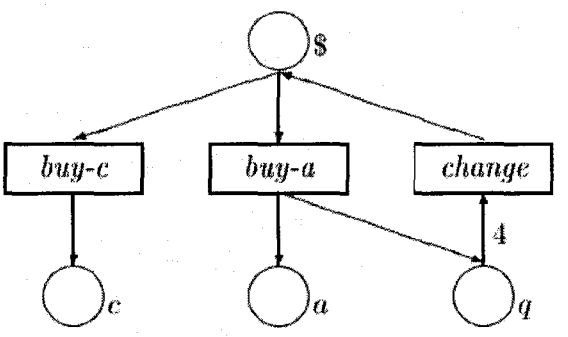


FIG. 3.4 – Un Petri Net

La figure 3.5 montre un court programme Maude défini à l'aide d'un module système. Il consiste en la représentation Maude du système décrit à la figure 3.4 à l'aide d'un *Petri Net*, un des modèles de concurrence les plus simples. Le programme consiste en un automate représentant une machine distributrice très simple. Les entrées sont les suivantes :

- $ représente l'entrée d'une pièce de 1$ dans la machine ;
- c représente un bonbon ;
- a représente une pomme ;
- q représente une pièce de 25 cents.

Le système permet donc l'achat d'un bonbon à 1 $ et l'achat d'une pomme pour 0,75 $.

La machine fait aussi la monnaie : pour quatre pièces de 0,.25 $, elle retourne une pièce de 1 $.

```
mod PETRI-MACHINE is
  sort Marking .
  ops null $ c a q : -> Marking [ctor] .
  op _ _ : Marking Marking -> Marking
       [ctor assoc comm id: null] .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [chng] : q q q q => $ .
endm
```

FIG. 3.5 – Le module système *PETRI-MACHINE*

Modules orientés-objet

Les modules orientés-objet sont développés à l'aide du niveau Full Maude. Ils sont alors leur propre notation spécifiquement développée pour le paradigme objet. La figure 3.6 montre comment une classe est déclarée à l'aide d'un module Maude orienté-objet.

```
class ClassName | att1 : attType .
```

FIG. 3.6 – Déclaration d'une classe sous la forme d'un module OO

Par contre, la notation Core Maude étant plus simple et Maude plus facile à utiliser sous cette notation, les modules orientés-objet ne seront pas utilisés dans ce mémoire. Le principe de réduction d'un module orienté-objet Maude sous une forme de module système sera utilisée. Pour se faire, la notation de la figure 3.7 sera utilisée.

```
sort ClassName .
subsort ClassName < Cid .
op ClassName : -> ClassName .
op att1 :_ : attType -> Attribute .
```

FIG. 3.7 – Déclaration d'une classe sous la forme d'un module système

On remarque alors la façon utilisée pour déclarer une classe et ses attributs, à l'aide des mots clés Cid, l'identifiant général de toutes les classes de Maude et Attribute, le type de Maude représentant un attribut d'une classe.

3.4 Maude et Spécifications formelles

Selon Meseguer et son équipe [63, 22], Maude est particulièrement approprié pour spécifier formellement plusieurs logiciels. En effet, tel que discuté précédemment, ce langage est basé sur une logique très mathématique et dispose d'une forte sémantique. Ainsi, Maude satisfait au critère suivant : être basé sur les mathématiques que les langages de spécifications formelles ont tous en commun.

Enfin, Maude offre également la possibilité de valider les descriptions développées à l'aide de simulations. Ceci n'est pas toujours possible avec d'autres langages, comme par exemple *Object-Z*. L'environnement Maude est également très flexible par rapport aux simulations puisqu'il permet de sélectionner une configuration initiale personnalisée, ce qui a pour effet de permettre la vérification d'une partie du système sans toutefois compromettre ce qui a été accompli auparavant.

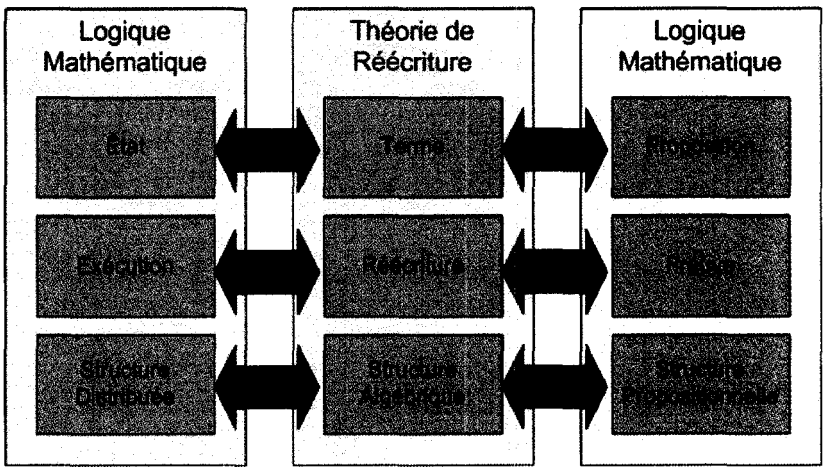


FIG. 3.8 – Parallèle entre un programme informatique, une théorie de réécriture et la logique mathématique

De fait, un programme Maude doit être vu comme un énoncé d'une théorie de réécriture. Ainsi, l'exécution de ce programme doit être interprétée comme une déduction logique à partir des axiomes définis dans le programme. La figure 3.8 présente les équivalences qui existent entre l'exécution d'un programme informatique, une théorie de réécriture et la logique mathématique.

3.5 Maude et Vérification de modèles

En plus d'être un langage simple, expressif et performant, Maude offre à ses usagers un vérificateur de modèle intégré à son environnement.

Du point de vue de l'environnement Maude, deux éléments sont nécessaires à l'utilisation du vérificateur de modèles [22]. Pour un module système Maude donné, ces deux éléments sont :

1. Une **spécification système** donnée par la théorie de réécriture décrite au sein de ce module ;
2. Une **spécification de besoins** donnée par une propriété ou une série de propriétés à propos du système en modélisation.

Bien évidemment, Maude supporte implicitement ces deux niveaux. En ce qui concerne le premier élément, de plus amples détails ont été donnés précédemment. Enfin, la spécification de besoins, au niveau de Maude, se fait à l'aide de la logique temporelle linéaire (LTL).

3.5.1 LTL et Maude

Dans cette section, la Logique temporelle linéaire (LTL) sera discutée avec de plus amples détails. Les opérateurs primitifs de la LTL sont d'abord introduits, pour ensuite parler des autres opérateurs existants.

Définition 3.5 (Logique temporelle linéaire) *Soit l'ensemble AP des* propositions atomiques, *on définit une proposition de logique temporelle linéaire $LTL(AP)$ inductivement par :*

- ***Vrai :*** $\top \in LTL(AP)$;
- ***Propositions atomiques :*** *Si* $\rho \in AP$ *alors* $\rho \in LTL(AP)$;
- ***L'opérateur Suivant (Next) :*** *Si* $\rho \in LTL(AP)$, *alors* $\bigcirc\rho \in LTL(AP)$;
- ***L'opérateur Jusqu'à (Untill) :*** *Si* $\rho, \phi \in LTL(AP)$, *alors* $\rho \mathcal{U} \phi \in LTL(AP)$;
- ***Connecteurs booléens :*** *Si* $\rho, \phi \in LTL(AP)$, *alors les formules* $\neg\rho$, *et* $\rho \vee \phi$ *sont aussi dans* $LTL(AP)$.

Bien sûr, plusieurs autres connecteurs booléens existent (voir la definition 3.6). Ces derniers peuvent être définis en fonction de l'ensemble minimal de la définition 3.5.

Définition 3.6 (Autres opérateurs booléens) *Les opérateurs suivants sont définis par rapport à l'ensemble minimal des opérateurs de la définition 3.5 :*

- ***Faux :*** $\bot = \neg\top$;
- ***Conjonction :*** $\rho \wedge \phi = \neg((\neg\rho) \vee (\neg\phi))$;
- ***Implication :*** $\rho \rightarrow \phi = (\neg\rho) \vee \phi$.

Enfin, puisque la LTL est une logique de type temporelle, elle nécessite l'utilisation d'opérateurs temporels. La définition 3.7 définit les opérateurs temporels de la LTL.

Définition 3.7 (Opérateurs temporels) *Les opérateurs temporels de la logique temporelle sont :*

- ***Éventuellement :*** *En anglais,* Eventually, *il est défini par* $\Diamond\rho = \top\mathcal{U}\rho$;
- ***Dorénavant :*** *En anglais,* Henceforth, *il est défini par* $\Box\rho = \neg\Diamond\neg\rho$;
- ***Libérer :*** *En anglais,* Release, *il est défini par* $\rho\mathcal{R}\phi = \neg((\neg\rho)\mathcal{U}(\neg\phi))$;
- ***Sauf si :*** *En anglais,* Unless, *il est défini par* $\rho\mathcal{W}\phi = (\rho\mathcal{U}\phi) \vee (\Box\rho)$;
- ***Entraîne :*** *En anglais,* Leads to, *il est défini par* $\rho \rightsquigarrow \phi = \Box(\rho \rightarrow (\Diamond\phi))$;
- ***Implication forte :*** *En anglais,* Strong Equivalence, *il est défini par* $\rho \Rightarrow \phi = \Box(\rho \rightarrow \phi)$;
- ***Équivalence forte :*** *En anglais,* Strong Equivalence, *il est défini par* $\rho \Leftrightarrow \phi = \Box(\rho \leftrightarrow \phi)$.

TAB. 3.1 – Opérateurs de LTL et notation Maude

Opérateur LTL	Notation Maude	Opérateur LTL	Notation Maude
$\top$	`True`	$\rho \rightarrow \phi$	`_->_`
$\bot$	`False`	$\rho \leftrightarrow \phi$	`_<->_`
$\neg\rho$	`~_`	$\Diamond\rho$	`<>_`
$\rho \wedge \phi$	`_/\_`	$\Box\rho$	`[]_`
$\rho \vee \phi$	`_\/_`	$\rho \mathcal{W} \phi$	`_W_`
$\bigcirc\rho$	`O_`	$\rho \rightsquigarrow \phi$	`_|->_`
$\rho \mathcal{U} \phi$	`_U_`	$\rho \Rightarrow \phi$	`_=>_`
$\rho \mathcal{R} \phi$	`_R_`	$\rho \Leftrightarrow \phi$	`_<=>_`

Les définitions précédentes résument les opérateurs de la LTL, et informent en même temps sur l'utilisation que l'on peut faire d'une telle logique. La forme utilisée pour présenter les divers opérateurs est une notation mathématique. Par contre, l'utilisation de ces opérateurs avec l'environnement Maude requiert une notation particulière. Le tableau 3.1 présente chacun des opérateurs vus précédemment et leur forme respective en Maude. Le symbole "_" représente l'emplacement d'une proposition atomique de LTL.

3.5.2 Structure de Kripke et Logique de Réécriture

Dans la section précédente, la liste complète des opérateurs temporels de la logique temporelle linéaire (LTL) ont été introduits. Il est alors important de noter qu'un modèle exprimé en LTL peut être vu comme étant une structure de Kripke. Essentiellement une *Structure de Kripke* est un système de transition non nommé total sur lequel un ensemble de prédicats unaires a été ajouté à son ensemble d'états. Ces quelques éléments discutés, la définition 3.8 introduit ce qu'est une structure de Kripke.

Définition 3.8 (Structure de Kripke) *Une Structure de Kripke est un triplet* $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ *où :*

- A *est un ensemble nommé* l'ensemble des états ;
- $\rightarrow_{\mathcal{A}}$ *est une relation binaire totale sur* A *nommée la* relation de transition ;
- *et enfin* $L : A \rightarrow \mathcal{P}(AP)$ *est une fonction qui, pour chaque état* $a \in A$, *associe l'ensemble* $L(A)$ *des propositions atomiques de* AP *qui sont vraies pour* a.

La sémantique de la logique temporelle linéaire (LTL) est définie à l'aide d'une *relation de Satisfaction* définie par : $\mathcal{A}, a \models \rho$.

Ceci signifie que la relation de satisfaction est définie entre une structure de Kripke $\mathcal{A}$ qui a l'ensemble AP comme propositions atomiques, un état $a \in A$ et une formule de logique temporelle LTL $\rho \in LTL(AP)$.

La relation de satisfaction des chemins peut être définie inductivement. Par contre, l'étude de cette induction mathématique dépasse le cadre de l'étude de ce mémoire.

Outre ces définitions théoriques, comment peut-on associer pratiquement une structure de Kripke à une théorie de réécriture $\mathcal{R} = (\Sigma, E, \phi, R)$ spécifiée par un module système Maude ? Il s'agit de spécifier les deux choses suivantes :

1. La sorte (*kind*) k sélectionnée de la signature Σ ;
2. Les prédicats d'états pertinents, ou plus spécifiquement l'ensemble AP pertinent des propositions atomiques.

Ainsi, il sera opportun de définir une série de prédicats à propos du système décrit à l'aide d'un module système Maude *M*. Par contre, il est plus ou moins opportun de définir ces prédicats à même ce module *M*. Il sera préférable de les définir dans un autre module spécifiquement prévu à cet effet.

Cette étape de définition de prédicats marque le début d'un processus bien défini à suivre pour utiliser efficacement le vérificateur intégré à l'environnement Maude. La section suivante présente dans le détail ce processus.

3.5.3 Vérification de Modèles avec Maude

Tel qu'il a été mentionné à la section précédente, il est possible d'associer une structure de Kripke à une théorie de réécriture développée pour représenter un système. Sachant que

le module *M* contient cette théorie de réécriture, donc représente le système, il s'agira de déterminer la sorte (*kind*) de la signature Σ et de développer une série de prédicats sur les états de ce système pour compléter l'association.

```
fmod LTL is
  sorts Prop Formula .
  subsort Prop < Formula .
*** primitive LTL operators
  ops True False : -> Formula [ctor format (g o)] .
  op ~_ : Formula -> Formula [ctor prec 53 format (r o d)] .
  op _/\_ : Formula Formula -> Formula
          [comm ctor gather (E e) prec 55 format (d r o d)] .
  op _\/_ : Formula Formula -> Formula
          [comm ctor gather (E e) prec 59 format (d r o d)] .
  op O_ : Formula -> Formula [ctor prec 53 format (r o d)] .
  op _U_ : Formula Formula -> Formula
          [ctor prec 63 format (d r o d)] .
  op _R_ : Formula Formula -> Formula
          [ctor prec 63 format (d r o d)] .
*** defined LTL operators
  op _->_ : Formula Formula -> Formula
          [gather (e E) prec 65 format (d r o d)] .
  op _<->_ : Formula Formula -> Formula
          [prec 65 format (d r o d)] .
  op <>_ : Formula -> Formula [prec 53 format (r o d)] .
  op []_ : Formula -> Formula [prec 53 format (r d o d)] .
  op _W_ : Formula Formula -> Formula
          [prec 63 format (d r o d)] .
  op _|->_ : Formula Formula -> Formula
          [prec 63 format (d r o d)] .
  op _=>_ : Formula Formula -> Formula
          [gather (e E) prec 65 format (d r o d)] .
  op _<=>_ : Formula Formula -> Formula
          [prec 65 format (d r o d)] .
...
endfm
```

FIG. 3.9 – Le module *LTL* de Maude

Il ne sera pas approprié de définir ces deux éléments à même le module *M*. Il sera plus approprié de le faire dans un autre module spécifiquement à cet effet. Les auteurs du manuel

de l'usager de Maude 2.0 [22] recommandent plutôt l'approche présentée ici. D'abord et avant tout, la figure 3.9 présente le module *LTL* de Maude, définissant tous les opérateurs de la logique LTL discutés à la section précédente. Le module a été tronqué après l'introduction des opérateurs primitifs et des opérateurs définis de la LTL.

Dans la section précédente, il a également été question d'une relation de Satisfaction. Elle faisait le lien entre une structure de Kripke $\mathcal{A}$, un état a et une formule de logique temporelle ρ. Le module *SATISFACTION* de Maude introduit l'opérateur $\models$ utilisé pour représenter cette relation (figure 3.10).

```
fmod SATISFACTION is
 protecting LTL .
 sort State .
 op _|=_ : State Formula ~> Bool .
endfm
```

FIG. 3.10 – Le module *SATISFACTION* de Maude

Les quelques éléments précédents sont en réalité des modules prédéfinis qui sont livrés avec l'environnement Maude. Ils ont cependant leur importance, et c'est pour cette raison qu'ils ont été expliqués ici. Dans ce qui suit, l'approche générique pour utiliser efficacement le vérificateur de modèles de Maude est démontrée plus en détail. Cette approche constitue une série d'étapes recommandées à accomplir pour utiliser adéquatement le vérificateur de modèles.

La toute première étape à franchir est la définition d'un module qui permettra de faire le lien entre une structure de Kripke et une théorie de réécriture. La structure de Kripke sera définie par les propriétés de LTL face au système, tandis que la théorie de réécriture est décrite par le module *M*. Un module *M-PREDS* est alors introduit pour créer le lien. Tel qu'il avait été mentionné précédemment, deux éléments sont nécessaires pour que le lien entre une structure de Kripke et une théorie de réécriture soit valide.

Le module *M-PREDS* est précisément utilisé pour définir ces éléments. La figure 3.11 montre ce module de façon générique. La ligne *subsort Configuration < State .* défini la sort (*kind*) de la signature qui sera utilisée au sein de la structure de Kripke. Enfin, le reste du

```
mod M-PREDS is
  protecting M .
  including SATISFACTION .
  subsort Configuration < State .
  ...
endm
```

FIG. 3.11 – Le module *M-PREDS* de Maude

module servira à introduire une série de prédicats sur les états du système.

L'exécution de la tâche de vérification avec Maude passe par la définition d'un dernier module. Ce dernier servira à la définition des états initiaux par où devront débuter les vérifications. Ceci simplifiera l'écriture des appels à la fonction de vérification de modèles de Maude immédiatement après, plutôt que d'écrire la configuration complète du système en entier. La figure 3.12 présente la structure type de ce module.

```
mod M-CHECK is
  including M-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER .
  ...
endm
```

FIG. 3.12 – Le module *M-CHECK* de Maude

Enfin, la dernière étape pour la vérification de modèles à l'aide de Maude se fait par l'appel à une fonction nommée *modelCheck* du module *MODEL-CHECKER*. Une partie de ce module est présentée à la figure 3.13. Un appel type à cette fonction y est également montré. De cet appel, on reconnaît les deux éléments suivants :

1. *initial_state* est l'état initial où est démarré la vérification. Le nom de cet élément a été introduit au module *M-CHECK* ;
2. *LTL_property* est une propriété de logique temporelle LTL définie à propos du système. C'est cette propriété que l'on tente de vérifier au sein du système *M*.

```
fmod MODEL-CHECKER is
  including SATISFACTION .  ...
  op counterexample : TransitionList TransitionList ->
     ModelCheckResult [ctor] .
  op modelCheck : State Formula ~> ModelCheckResult .
  ...
endfm

modelCheck(initial_state, LTL_Property) .
```

FIG. 3.13 – Une partie du module *MODEL-CHECKER* de Maude et un appel type à la fonction *modelCheck*.

3.6 Choix de Maude

Aux premiers abords, l'environnement Maude est relativement nouveau. La version 2.0 de l'environnement ayant été rendue publique en 2005, et sachant que l'environnement a connu quelques autres évolutions (les versions 2.1 et 2.2 respectivement), ceci suggère que cet outil est encore en voie de développement.

De plus, comme l'environnement est relativement nouveau, un nombre restreint d'auteurs s'y sont intéressés jusqu'à maintenant. Il était donc intéressant d'explorer les possibilités de ce nouvel outil face aux SOO. De plus, l'environnement Maude, basé sur la logique de réécriture, a été spécialement développé pour supporter la programmation et la spécification de systèmes concurrentiels. Il s'avérait également tout aussi intéressant d'explorer les possibilités de Maude par rapport à cet aspect.

Le chapitre 1 a présenté plusieurs approches pour la spécification formelle de systèmes. Lui aussi basé sur de solides fondements mathématiques, Maude est un candidat idéal pour la spécification de système. Comparativement à des langages tels *Object-Z* ou *B*, Maude aura le net avantage de permettre la définition de ses propres notations (Maude est un méta langage). Ceci rend Maude beaucoup plus expressif et beaucoup plus aisé à comprendre comparativement aux notations abstraites et mathématiques d'*Object-Z* et autres.

D'un autre côté, tel que mentionné au chapitre 2, il existe sur le marché de très nombreux outils pour la vérification de modèles. Certains sont plus connus que d'autres, plus populaires,

etc. De ceux-là, mentionnons SMV et SPIN.

Très certainement, SPIN est le vérificateur le plus répandu et le plus utilisé. Plusieurs approches ont par ailleurs développé des outils permettant de traduire une notation quelconque vers le langage Promela, le langage d'entrée du vérificateur SPIN.

Un peu comme l'était *Object-Z* du côté des spécifications formelles, Promela a ses propres notations. Cela nécessite cependant un certain apprentissage avant de pouvoir l'utiliser efficacement.

Encore une fois, Maude offre la possibilité de définir des opérateurs propres à l'usager, rendant les programmes beaucoup plus lisibles et compréhensibles. Maude a aussi l'avantage d'être un langage de programmation proprement dit, comparativement aux langages de spécification formelle et de vérification. Ces derniers sont très souvent spécifiques de ce type de technique, obligeant ainsi l'usager à apprendre un langage de spécification ou de vérification avant de pouvoir bénéficier des avantages de chacune de ces techniques. Il sera également obligé de connaître un langage de programmation pour pouvoir ensuite développer son système.

Les notations Maude comblent donc ces deux aspects dans un seul et unique langage. Une notation Maude servira autant de spécification formelle (et à la vérification de modèles par le fait même), que de langage de programmation permettant le développement du produit. Un usager n'aura alors qu'à apprendre un seul langage, Maude, et pourra programmer son application tout en bénéficiant d'outils de vérification avancés directement dans son environnement de travail.

Toutes ces raisons ont motivé le choix de Maude comme environnement de travail pour le développement d'une nouvelle méthodologie de vérification formelle de diagrammes UML. Le chapitre 4 introduira et discutera cette approche.

Chapitre 4

Vérification Formelle de Diagrammes UML

Tel qu'étudié aux chapitres 1 et 2, les techniques de spécification formelle et de vérification formelle offrent chacune plusieurs avantages considérables lors du développement de logiciels.

L'environnement Maude, construit spécifiquement pour les systèmes concurrentiels et offrant un cadre formel adéquat pour spécifier formellement un système logiciel, constitue un candidat parfait pour la translation de diagrammes UML vers une notation formelle. De plus, le vérificateur de modèles intégré à cet environnement permettra par la suite la vérification formelle des modèles qui seront développés.

Dans les pages qui suivent, un bref rappel sera fait d'abord sur le paradigme de programmation concurrentielle, qui est un des paradigmes les plus répandus, ainsi que sur le langage UML, plus particulièrement sur les types de diagrammes utilisés. Par la suite, un processus de translation sera présenté. Ce processus a pour objectif de translater les diagrammes UML considérés pour modéliser un système en développement vers des notations formelles Maude. La présentation du procédé utilisé en vue de la vérification formelle des modèles développés viendra par la suite.

4.1 Rappels

Avant d'introduire l'approche adoptée pour la vérification de modèles, il est important de rappeler quelques notions importantes.

Tout d'abord, les systèmes concurrentiels constituent aujourd'hui un groupe important des systèmes développés. Comme l'approche proposée sera testée sur un système de ce type, la section 4.1.1 introduira quelques notions de base relatives à ces systèmes.

Par la suite, un bref rappel sur les notations UML sera fait par rapport aux trois types de diagrammes qu'utilisera la méthode proposée : les diagrammes de classes, les diagrammes d'états-transitions et les diagrammes de communication.

4.1.1 Rappel sur les systèmes concurrentiels

De nos jours, il existe de nombreux paradigmes de programmation tous aussi diversifiés les uns que les autres. Nombre d'entre eux ont été développés avec pour objectif l'utilisation de techniques diverses.

Parmi ces paradigmes de programmation, l'un des plus répandus est très certainement la programmation concurrentielle [33]. Ce paradigme a essentiellement pour objectif de permettre l'accomplissement des diverses tâches dévolues à un système de façon simultanée : les tâches sont accomplies en même temps, de façon indépendantes les unes des autres, sauf dans les cas où ces tâches doivent communiquer et / ou partager une même ressource [33, 35].

De par leur nature, les objets sont particulièrement intéressants pour être adaptés à la programmation concurrentielle. Ainsi, les objects seront les parfaits candidats à la concurrence. Il suffira d'associer par exemple un objet à un processus système, et ainsi chaque objet aura son propre environnement d'exécution. Ces objets pourront donc accomplir leur tâches indépendamment des autres. Ce type de programmation est assez complexe mais offre néanmoins de nombreux avantages. En plus d'accomplir un travail plus rapidement, l'utilisation des possibilités d'un micro-ordinateur sont maximisées.

Cependant, ce type de programmation souffre également de problèmes qui lui sont spécifiques. Les boucles infinies, les interblocages et les inconsistences au niveau des données,

pour ne citer que ceux là, peuvent aisément survenir [33, 24]. Une boucle infinie survient lorsqu'un chemin d'exécution tente d'exécuter le même bout de code, et ne parvient pas à satisfaire la condition de sortie d'une boucle. Un interblocage survient quant à lui lorsque, par exemple, deux processus concurrents attendent tous les deux la libération d'une ressource qu'ils partagent, et qu'aucun d'eux ne sort de ce mode d'attente lorsque la ressource est libre. Enfin, des inconsistances au niveau des données pourraient survenir lorsque deux processus concurrents accèdent (de façon non contrôlée) à la même ressource en même temps. L'un d'eux écrit de nouvelles données, alors que le second y accède en lecture croyant accéder aux nouvelles informations que le premier n'a pas encore terminé de mettre à jour.

Wegner, dans [87], introduit plusieurs concepts pertinents aux objets concurrentiels. Tout d'abord, il introduit le concept d'object actif (*Active Object*). Contrairement aux objects habituels, qui sont activés à la réception d'un message, ce type d'objet peut déjà être actif lors de la réception d'un message. Ainsi, ces objets ont à leur disposition un tampon de réception. Les messages reçus seront alors exécutés de façon séquentielle selon leur ordre de réception.

Wegner parle aussi de concurrence interne et de concurrence externe pour les objets [87].

La concurrence interne d'un objet réfère à un objet montrant un comportement individuel accomplissant plusieurs tâches en même temps ;

La concurrence externe entre objets réfère, quant à elle, au caractère indépendant de deux objets actifs.

Enfin, Wegner introduit trois termes permettant de qualifier la concurrence d'un système : la concurrence externe, la quasi concurrence et la concurrence totale. Ces termes sont définis par ce qui suit.

La concurrence externe : La concurrence externe fait référence à deux objets (ou plus) tentant d'accéder simultanément à une ressource commune ;

La quasi concurrence : Un objet actif quasi concurrent est un objet pour lequel le comportement interne montre des caractéristiques concurrentes ;

La concurrence totale : Un objet actif concurrent disposera, quant à lui, de plusieurs processus d'exécution.

4.1.2 Rappel sur UML

UML (*Unified Modeling Language*) est un langage offrant de nombreux artéfacts en vue de la construction d'un système logiciel. Avec ses vues et ses divers types de diagrammes, UML offre la possibilité de modéliser la structure statique d'un système en plus de son comportement collectif à différents niveaux (individuel, collaboratif, global).

En ce qui concerne l'approche présentée dans ce mémoire, on considère trois types de diagrammes :

Les diagrammes de Classes : Diagrammes qui modélisent la structure statique d'un système ;

Les diagrammes d'États-Transitions : Diagrammes qui modélisent le comportement interne des classes (comportement individuel) ;

Les diagrammes de Communication : Diagrammes qui modélisent le comportement collaboratif de plusieurs objets en vue de la réalisation d'une tâche spécifique.

Dans ce qui suit, plus d'informations sont présentées à propos de chacun de ces types de diagrammes.

Diagrammes de Classes UML

D'une manière générale, les *Diagrammes de Classes UML* expriment l'organisation structurelle d'un système logiciel orienté-objet [68, 57]. La structure d'un système décrite par ce type de diagramme est donnée en termes de classes, ainsi que par les relations qui existent entre elles. Il y a plusieurs types de relations entre les classes, dont les principaux sont les suivants :

L'Agrégation : L'Agrégation est une sorte de relation représentant la composition. Il en existe deux types, l'agrégation (exprimée avec un losange blanc), représente un lien n'ayant pas d'impact sur l'existence d'un objet, tandis que le lien de Composition (losange noir) représente quant à lui un lien ayant un impact sur l'existence d'un objet (par exemple, un "objet" main est composé de 5 "objets" doigts) ;

L'Héritage : Souvent appelé lien de Généralisation – Spécialisation, ce type de relation met en relation une ou plusieurs classes avec une première, nommée classe mère, comportant des caractéristiques communes à tous ses enfants, autant du point de vue structurel que comportemental ;

L'Association : Le lien d'Association représente le fait qu'il existe un lien dynamique entre deux objets des classes respectivement liées. Ce lien a une quelconque utilité face au domaine.

Au même titre qu'une classe décrit un ensemble d'objets, une association, quant à elle, décrit un ensemble de liens. Les divers objets impliqués dans un système sont des instances des classes, tandis que les liens qui existent entre divers objets sont des instances des associations.

Un diagramme de classe n'exprime rien de particulier sur les liens d'un objet donné, mais décrit plutôt de façon abstraite les liens potentiels d'un objet quelconque vers d'autres objets [68]. La figure 4.1 montre un exemple de Diagramme de Classes UML.

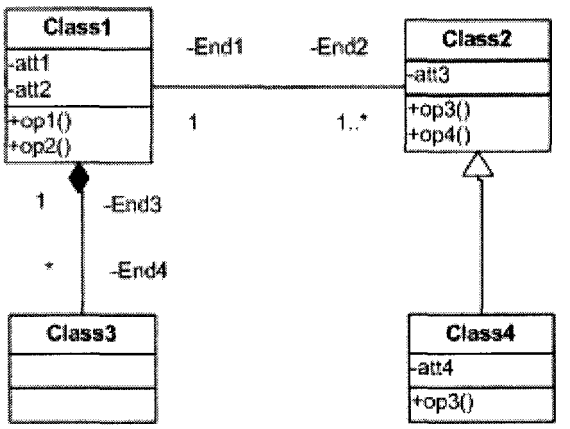


FIG. 4.1 – Exemple de Diagramme de Classes UML

Dans la figure précédente, on reconnaît la plupart des éléments discutés précédemment. Tout d'abord, on peut remarquer la structure de chaque classe, avec ses attributs et ses opérations. Un lien de chaque type est inséré dans ce diagramme afin de les illustrer. Le lien entre les classes *Class1* et *Class2* est de type *Association*. Quant à lui, le lien entre les classes

Class1 et *Class3* est une relation d'*Agrégation*, ou plus précisément une *Composition*. Enfin, le lien entre les classes *Class2* et *Class4* est un lien d'*Héritage*.

Diagrammes d'États-Transitions UML

Les diagrammes d'états-transitions UML [68] sont en réalité des automates à états finis. Mathématiquement parlant, ce sont des graphes orientés. Leur rôle principal est de décrire le cycle de vie des objets, les événements qu'ils subissent, leurs transitions et leurs états. Un diagramme d'états-transitions est composé de plusieurs éléments : les états (ou noeuds du graphe), ainsi que les événements et les transitions (les arcs du graphe).

On pourrait définir un état par la condition dans laquelle se trouve un objet entre les divers messages qu'il reçoit. Un état se définit par deux choses différentes : par la valeur des attributs de l'objet et par la valeur des liens avec les autres objets à un instant donné. La valeur des attributs est critique ici, et particulièrement lorsque mise en contexte avec les divers liens qui existent avec d'autres objets. Toutes ces données mises ensemble peuvent avoir une influence sur la réaction qu'aura l'objet face au prochain message qu'il recevra de l'extérieur. Avec UML, on représente un état par une boîte aux coins arrondis. Un diagramme d'états-transitions est habituellement constitué d'un état initial, d'états intermédiaires (zéro ou plusieurs) et de un ou plusieurs états finaux (zéro ou plusieurs).

Le second concept d'importance face aux diagrammes d'états est la transition. Une transition est le lien qui existe entre deux différents états. C'est par ce lien que l'objet peut changer d'état. Lorsqu'un événement se produit (nous verrons dans ce qui suit ce qu'est un événement), il peut entraîner un changement d'état face à un objet, mais conditionnellement à ce que l'événement en question ne soit associé à une transition. Notons qu'il est impossible de transiter entre un état donné et un autre sans qu'une transition existe entre les deux. Avec UML, une transition est représentée par une flèche entre deux états. Par ailleurs, une transition peut être gardée. Les gardes sont évaluées dès l'occurrence de l'événement de déclenchement. Elles permettent de maintenir l'aspect déterministe d'un automate d'états finis, même lorsque le même événement est susceptible de déclencher plusieurs transitions.

Les événements, quant à eux, pourraient être définis par quelque chose de remarquable par

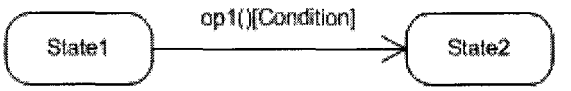


FIG. 4.2 – Exemple d'un événement déclenchant une transition

un objet. Cet événement se produit à un instant quelconque lors de l'exécution du système. La figure 4.2 montre un exemple d'un événement déclenchant une transition. Cet événement peut provenir du système ou de son interface, et peut être de trois natures différentes. Un événement dit externe, ou événement système, est un événement causé par un des acteurs. Un événement dit interne est un événement causé par une entité du système même. On pourrait citer comme exemple l'invocation d'une méthode de l'objet par l'envoi d'un message par un autre objet. Enfin, un événement temporel et lié à un temps écoulé ou à une date et une heure spécifique. Par exemple, on pourrait faire déclencher une méthode quelconque après qu'un laps de temps déterminé ce soit écoulé.

Enfin, les diagrammes d'états peuvent êtres utilisés pour modéliser la concurrence interne d'un objet [8, 9, 79]. En effet, il existe une notation permettant d'expliciter la résolution concurrentielle de diverses tâches dévolues à un même objet. Ce type de notation passe par un état composite à régions orthogonales concurrentes. La figure 4.3 présente un état composite à régions orthogonales concurrentes génériques.

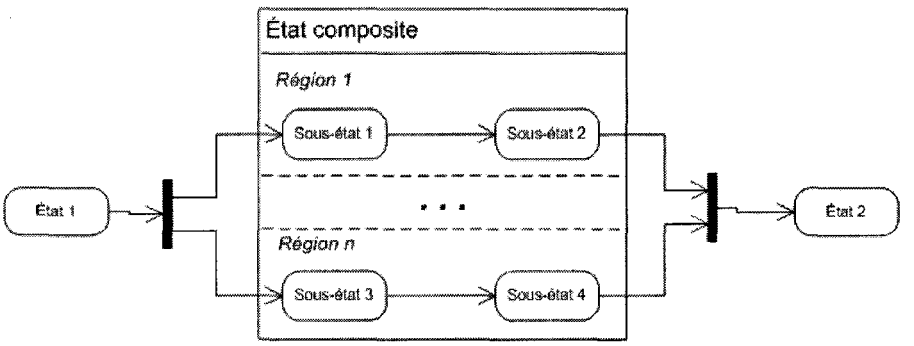


FIG. 4.3 – Exemple d'un état composite à régions orthogonales concurrentes

La figure 4.3 présente plusieurs éléments importants. Tout d'abord, l'état nommé *État composite* est la structure centrale d'un diagramme d'état concurrentiel. Cet état, tel que

mentionné précédemment, montre plusieurs régions orthogonales différentes, séparées par une ligne hachurée. Chacune de ces régions représente le comportement de l'objet concernant une tâche à accomplir. Aussi, chacune de ces régions est exécutée indépendamment des autres régions. L'entrée dans ces régions se fait à l'aide d'un point d'entrée au diagramme, tel qu'il est commun de le faire pour un diagramme habituel. L'entrée peut également se faire à l'aide d'un point de séparation (une structure *fork*), tel que le montre la figure 4.3. À ce moment, la réunion des parties concurrentes se fera à l'aide d'une structure *Join*.

Diagrammes de Communication UML

Les diagrammes de collaboration UML [77, 68, 58, 71], dorénavant connus sous le nom de diagrammes de communication avec la version 2.0 d'UML [72] représentent une vue dynamique du système. Ils décrivent comment un ensemble d'objets collaborent pour la réalisation d'une tâche particulière, ainsi que les liens entre ces objets. Les diagrammes de communication montrent les interactions dynamiques entre ces objets à travers la représentation d'envoi de messages. Ils insistent particulièrement sur la structure spatiale qui permet la mise en communication d'un groupe d'objets. L'ajout d'une dimension temporelle requiert la définition de numéros de séquence pour les messages (séquencement des messages). Un message est la matérialisation d'une communication au cours de laquelle se transmettent des informations et qui permet éventuellement d'obtenir des résultats. Un envoi de message entre un émetteur (source) et un récepteur (destinataire) nécessite que le récepteur puisse réaliser l'activité définie par le message. Par exemple, dans le cas d'un appel de méthode, l'opération invoquée doit être définie dans le récepteur et posséder une visibilité appropriée. Un envoi de message peut être synchrone (représenté par une flèche à extrémité triangulaire), ou asynchrone (représenté par une demi-flèche) comme l'illustre la figure 4.4.

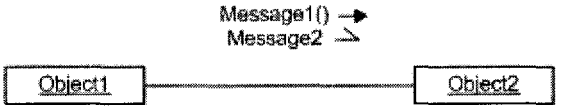


FIG. 4.4 – Message synchrone (*Message1*) et message asynchrone (*Message2*)

Les messages échangés entre les objets au sein d'un diagramme de communication (voir figure 4.5) peuvent être envoyés de diverses façons. La première façon est séquentielle (les messages sont soit de même niveau et sont alors incrémentés (1, 2, 3, ...), soit déclenchés par un autre message (comme c'est le cas pour les appels des méthodes emboîtées dans un appel englobant)). La deuxième façon est concurrente (deux messages concurrents ont un numéro de séquence identique qui diffère uniquement par le nom adjoint (par exemple 2a et 2b sont concurrents)). Les deux façons peuvent être utilisées à la fois. Le concept de synchronisation est aussi utilisé dans le cas où l'envoi d'un message nécessiterait l'envoi d'une liste de messages concurrents. Cette dernière possibilité est connue sous le nom de *Point de synchronisation d'un message*. Ce point de synchronisation est exprimé sous la forme d'une séquence d'envois de messages terminée par le caractère "/". En plus, chaque envoi de message peut être conditionnel ou non.

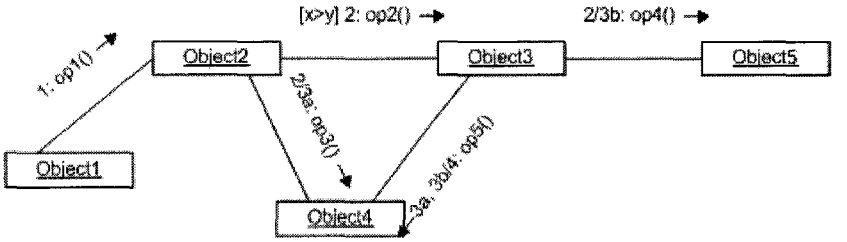


FIG. 4.5 – Exemple de diagramme de communication

L'exemple donné par la figure 4.5 illustre les différentes possibilités citées ci-dessus. Les messages *op1* et *op2* sont envoyés de façon séquentielle. Les deux messages *op3* et *op4* sont exécutés en concurrence, après le message *op2*. L'envoi du message *op5* nécessite les envois des messages *op3* et *op4*.

Enfin, lorsque des objets disposent de leur propre processus d'exécution, ils sont nommés objets actifs (*Active Objects*). Dans ce cas, l'identificateur d'un objet de ce type portera la mention *active* au sein du diagramme de communication.

4.1.3 UML et Incohérences

En utilisant ces différents diagrammes, UML offre la possibilité de décrire plusieurs "vues" d'un même système. Cependant, ces différentes vues peuvent être en conflit puisqu'elles peuvent présenter des incohérences [1]. De plus, l'expérience montre que de très nombreuses fautes de modélisation sont perceptibles à travers la détection d'incohérences [69, 76, 66, 4]. Une incohérence est la violation d'une propriété associée au langage UML qui doit être respectée par tout modèle UML [56]. En effet, la cohérence peut concerner un seul diagramme (on parle de cohérence intra-diagramme) ou plusieurs diagrammes (on parle de cohérence inter-diagrammes). Parmi les incohérences intra-diagramme, on peut mentionner la présence de cycles dans un graphe d'héritage pour un diagramme de classes, l'existence d'une transition dont l'état source est un état final dans un diagramme d'états-transitions et le non respect de l'ordre d'envoi de message d'un point de synchronisation dans un diagramme de communication (par exemple, le numéro du message bloqué est inférieur au numéro du message bloquant). La représentation des systèmes complexes, en utilisant les différents diagrammes cités précédemment, peut engendrer des incohérences inter-diagrammes. On peut citer, entre autres, un événement appel d'un diagramme d'états-transitions qui ne figure pas comme méthode dans la classe correspondante. Aussi, un envoi de message (appel de procédure) dans un diagramme de communication ne figurant pas comme une méthode à visibilité appropriée dans la classe de l'objet destinataire. Par ailleurs, l'état final d'un système fini ne se compose pas de différents états finaux des différents objets impliqués dans le diagramme de communication.

4.2 Translation de diagrammes UML vers une spécification formelle Maude

Au chapitre 1, les concepts pertinents aux spécifications formelles ont été introduits. Visiblement, les avantages d'une telle technique sont nombreux, et offrent d'intéressantes possibilités lorsqu'elles sont utilisées pour le développement d'un système.

De plus, au chapitre 3, l'environnement Maude a été introduit. Ce langage, disposant d'une base mathématique solide, est un candidat parfait pour spécifier formellement un système. Outre le fait qu'il a été conçu avec cet objectif, Maude offre la possibilité de simuler le système développé, tel qu'il sera discuté plus loin dans cette section.

4.2.1 Translation de diagrammes UML vers Maude

Cette section présente une technique pour la translation de trois types de diagrammes UML en une notation formelle Maude. Les trois types de diagrammes utilisés sont les diagrammes qui ont été présentés dans les rappels : les diagrammes de classes, d'états-transitions et de communication.

La technique consiste à dériver systèmatiquement une description formelle Maude à partir de l'analyse de ces trois types de diagrammes. Le processus de translation est divisé en trois grandes étapes majeures, que l'on peut décrire comme suit. De plus, la figure 4.6 résume de façon visuelle ces mêmes étapes.

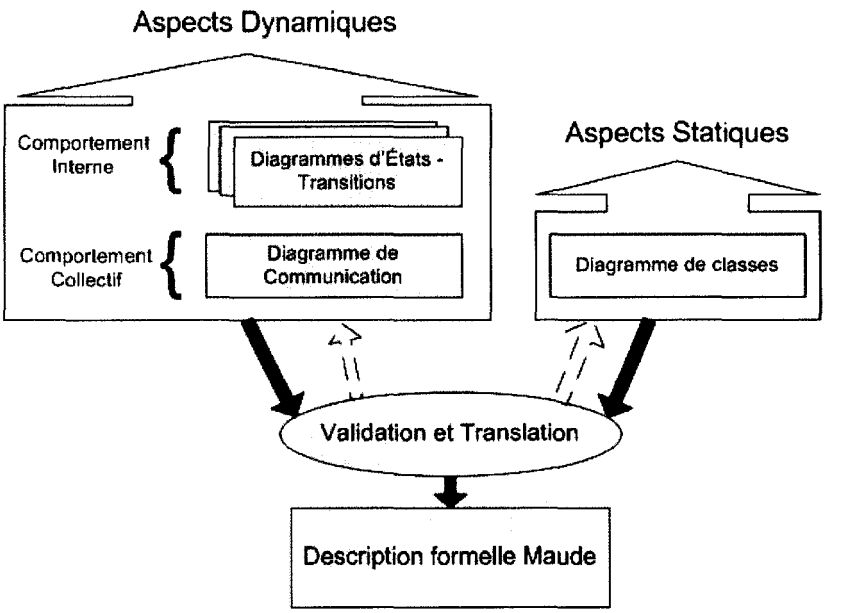


FIG. 4.6 – Aperçu du processus de translation

1. Décrire un système en développement à l'aide de diagrammes UML, plus particulièrement les trois types de diagrammes qui seront décrits formellement à l'aide de Maude ;
2. Vérification inter-diagrammes ;
3. Génération de la description formelle Maude.

La première étape du processus de translation est en réalité l'étape d'analyse habituelle du processus de développement d'un système logiciel. Elle consiste en la description d'un système à l'aide des artifices d'UML. Bien évidemment, de nombreux artifices UML existent, ainsi que de nombreux types de diagrammes. Ceci est d'autant plus vrai avec la version 2.0 d'UML [72]. En ce qui concerne l'approche proposée, elle prend en considération les trois types de diagrammes décrits précédemment.

Tel que discuté au chapitre 2, il est plutôt rare d'avoir une approche de spécification formelle (ou de vérification de modèles, d'ailleurs) qui considère à la fois les aspects statiques et les aspects dynamiques, autant individuel que collectif, des systèmes orientés-objet. L'approche proposée, quant à elle, considère tous ces aspects conjointement. Le fait de considérer tous ces aspects est le concept novateur de l'approche proposée dans ce mémoire.

Le diagramme de classes est utilisé pour représenter la structure statique du système, la structure de ces classes, etc. ;

Les diagrammes d'états-transitions de chacune des classes ont pour objectif de rendre compte du comportement individuel des objets de ces classes ;

Le diagramme de communication (anciennement, le diagramme de collaboration) a pour objectif de représenter le comportement collaboratif des objets impliqués dans l'accomplissement d'une tâche.

La seconde étape du processus de translation a pour objectif de vérifier, autant que possible, l'exactitude des diagrammes utilisés. Une vérification inter-diagrammes est effectuée sur tous les diagrammes utilisés pour s'assurer de la présence de tous les éléments nécessaires. Bien évidemment, les diagrammes UML, et particulièrement les diagrammes d'états-transitions, sont des structures très riches qui ont la possibilité de représenter de nombreux

éléments via leurs notations. L'approche proposée se limite donc, premièrement à ces trois types de diagrammes, et aux notations de base de ces mêmes diagrammes. Les notations supportées ont été décrites dans la section 4.1.2 des rappels sur les notations UML.

Quant à elle, la troisième étape du processus de translation consiste en la génération systématique de code source du langage Maude à partir des diagrammes UML considérés.

L'approche considère un diagramme de classes et un diagramme de communication à la fois, ainsi que tous les diagrammes d'états-transitions des classes impliquées dans la communication. Ainsi, la vérification s'effectue sur une tâche du système à la fois. Pour formaliser d'autres tâches, il s'agira de répéter le processus autant de fois que nécessaire.

Tel que mentionné au chapitre 3, la structure d'un programme Maude est divisé en modules de divers types. La génération systématique de code source Maude lors de l'application du processus de translation décrit précédemment se fera également au moyen de plusieurs modules Maude. Plusieurs modules seront ainsi générés. La figure 4.7 résume visuellement les modules qui seront générés.

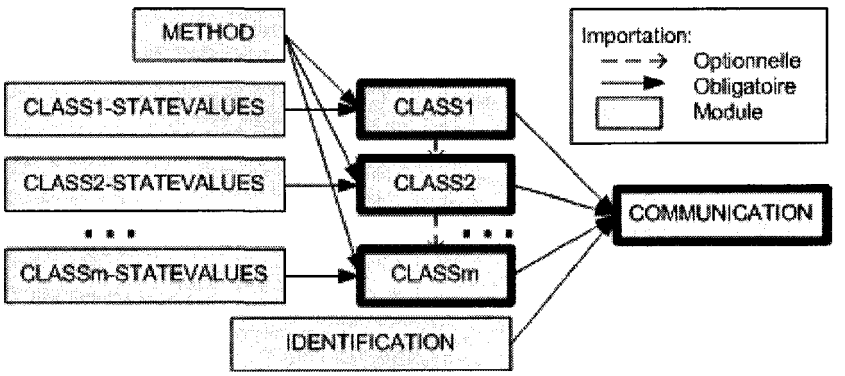


FIG. 4.7 – Modules générés par le processus de translation

Plusieurs modules sont donc générés lors de la translation systématique de diagrammes UML en une notation formelle Maude. De ces modules, tous ceux ayant un encadrement normal sont des modules fonctionnels de Maude, tels que décrits au chapitre 3. Les modules avec un cadre gras sont des modules système. Il aurait peut-être été préférable d'utiliser des modules orientés-objet pour ces derniers. Par contre, un module orienté-objet peut être

réduit à une notation système sans problème avec Maude, et ceci aura pour effet de simplifier les notations. En effet, l'utilisation de modules orientés-objet nécessite l'utilisation de *Full Maude*, alors que la notation système, équivalente, ne nécessite que *Core Maude*.

Les modules générés peuvent être décrits de la façon suivante. Le module *METHOD*, présenté à la figure 4.8, n'est pas développé en fonction des diagrammes UML considérés, mais est plutôt toujours le même. Il contient une série d'éléments pertinents à la translation, soit la définition des types nécessaires à la définition de paramètres et de types de retour pour les méthodes des objets.

```
fmod METHOD is
  sorts  ParameterList ResultType Parameter Void .
  subsort Parameter < ParameterList .
  subsort Void < ResultType .
  op EmptyParameterList : -> ParameterList .
  op _,_  : Parameter ParameterList -> ParameterList .
endfm
```

FIG. 4.8 – Le module *METHOD*

Dans ce module, quatre types différents sont introduits, et sont utilisés pour les fins suivantes :

ParameterList : sera utilisé comme opérateur générique pour la définition de la liste des paramètres d'une méthode ;

ResultType : sera utilisé comme opérateur générique pour la définition du type de retour d'une méthode ;

Parameter : est le type générique d'un paramètre d'une méthode ;

Void : est un type particulier de *ResultType*, comme en fait foi une ligne subséquente, et représente le fait qu'une méthode n'a pas de valeur de retour.

De plus, on introduit un nouvel opérateur, *EmptyParameterList*. Ce dernier servira à signifier l'occurrence où une méthode n'a pas de paramètres. Enfin, l'opération _,_ est un constructeur pour la liste de paramètres d'une méthode.

Les autres modules varieront selon les diagrammes utilisés. En premier lieu, un module fonctionnel est défini pour chacun des diagrammes d'états-transitions du système. Dans chacun des modules, un type est défini, ainsi qu'une série d'opérateurs. Ces opérateurs portent le nom des valeurs des états de la classe. Ainsi, pour chacun des diagrammes d'états-transitions, un module fonctionnel Maude lui est associé. Ce module porte le nom de la classe associée au diagramme concaténé à la chaîne de caractères "*STATEVALUES*".

Ensuite, pour chacune des classes du système, un module système est défini. Ce module porte le nom de la classe. L'élément principal de ce module est sans conteste la définition de la classe elle-même. La figure 4.9 montre la forme générique d'une classe. La notation présentée ici est orientée-objet (se référer au chapitre 3 pour la notation d'un module système).

```
class ClassName | State : ClassNameStateValues
        [, RegSubstateAtt] [, ComponentList]
        [, AttributeList] .
```

FIG. 4.9 – Définition générique d'une classe

Dans cette définition, on retrouve d'abord le nom de la classe, *ClassName*, un attribut *State* qui servira à décrire l'état dans lequel se trouve un objet, ainsi que trois listes de composants, définis par :

RegSubstateAtt représente une série d'attributs représentant l'état actuel des régions orthogonales d'un super état. Ceci est utilisé dans l'éventualité où le diagramme d'états-transitions d'une classe montrerait un état composite avec plusieurs régions orthogonales concurrentes. Pour chacune des régions, on introduit un attribut dans la liste *RegSubstateAtt*. Le fonctionnement de ces attributs est simple : lorsqu'une transition s'effectue dans une des régions, seul l'attribut correspondant est modifié. L'attribut *State* global, quant à lui, ne variera que lorsque l'objet quittera son état composite concurrent ;

ComponentList représente une série de composants de la classe, dans le cas où le diagramme de classes montre une relation d'agrégation. Pour chaque lien d'agrégation dans ce diagramme, on retrouvera un attribut du type de la classe agrégée, et le module de la

classe devra également importer le module de la classe agrégée ;

AttributeList représente une série d'attributs habituels de la classe. Afin de limiter le nombre d'attributs présents, on limitera ceux décrits ici aux attributs ayant un impact direct sur le comportement individuel et collectif de l'objet.

Chacun des modules décrivant une classe contiendra également la définition de chacune des méthodes d'une classe. Les méthodes seront définies à partir de la forme générique donnée à la figure 4.10, qui reprend les éléments introduits dans le module *METHOD* (voir la figure 4.8).

```
op FunctionName : ParamaterList -> ResultType .
```

FIG. 4.10 – Définition générique d'une méthode d'une classe

Le module *IDENTIFICATION* aura pour objectif de déclarer un mécanisme d'identification des objets adéquat. Les objets prenant part au comportement collectif décrit dans un diagramme de communication devront être identifiés d'une certaine façon. Pour ce faire, le module *IDENTIFICATION* déclare, pour chaque classe, un identificateur d'objets qui sera un sous-type du type *Oid*, le type des identificateurs d'objets général. On y déclarera également un identificateur d'objets *Receiver*, qui lui, sera un sous-type de chacun des types décrits auparavant. Cet identificateur sera utilisé dans la déclaration des messages passés entre les objets. Les informations pertinentes à ce sujet figurent un peu plus loin.

Le module *COMMUNICATION* est très certainement le module le plus important généré lors de la translation de diagrammes UML en une notation formelle Maude. Ce module importe d'abord tous les autres modules décrits jusqu'à maintenant, et aura pour objectif de les étendre en ajoutant des propriétés comportementales à l'aide de règles de réécriture. Ces règles de réécriture prendront à la fois en considération le comportement individuel des objets décrits à l'aide de diagrammes d'états-transitions, et le comportement collectif de ces derniers décrit à l'aide d'un diagramme de communication.

Afin d'accomplir la tâche qui leur est dévolue, les objets qui collaborent devront s'échanger divers messages entre eux. La structure d'un de ces messages est donnée à la figure 4.11,

ligne 1. L'interprétation de ce message se fait comme suit. Le destinataire du message est l'objet *Receiver* (voir le module *IDENTIFICATION* à ce sujet). *ResultType*, quant à lui, a été introduit dans le module *METHOD* (figure 4.8). Il provient donc de l'exécution d'une méthode d'un objet d'une certaine classe. La figure 4.10 le décrit plus adéquatement. Ainsi, un message destiné à un objet *X* demandant l'exécution du message *M* n'ayant aucun paramètre aura la forme *ComingMsg (M (EmptyParameterList), X)*.

```
op ComingMsg :  ResultType Receiver -> Msg .        ***[1]
op IsAccomplished :  ResultType Receiver -> Msg .   ***[2]
```

FIG. 4.11 – Forme générique des messages et des messages de synchronisation

Un peu plus tôt dans ce chapitre, le concept de point de synchronisation a été introduit. Il s'agit, par exemple, du fait qu'un message donné doit attendre que plusieurs autres messages aient tous été complétés avant de ne pouvoir débuter son exécution. Il faut donc introduire une nouvelle notation qui aura pour effet de signaler qu'un message a été complété avec succès. L'objectif du message de synchronisation *IsAccomplished* est d'accomplir ceci. Sa structure générique, donnée à la figure 4.11 ligne 2, est identique à celle d'un message conventionnel et s'interprète de la même façon, à l'exception qu'il faut comprendre que son exécution est complétée au moment où le message de synchronisation *IsAccomplished* apparaît.

Les règles de réécriture qui seront introduites dans le module *COMMUNICATION* auront pour configuration initiale et finale la configuration partielle du système global. Il est entendu par ceci qu'il y figurera un ou plusieurs objets notés sous la forme Maude d'un objet < *C : ClassName* | *StateC : ClassName_StateValue [, Attributes]* >, ainsi qu'une série de messages conventionnels et de synchronisation. On y retrouve les éléments habituels : *C* est un identificateur d'objet de classe *ClassName*. La classe comporte un attribut *StateC* de valeur actuelle *ClassName_StateValue* décrite au module *ClassNameStateValues*. La classe pourrait également comporter un nombre quelconque d'attributs généralisés par la notation *[, Attributes]*.

Les systèmes orientés-objet concurrentiels (SOOC), comme il a été décrit plus haut, sont dorénavant très répandus. Il serait alors intéressant que la méthode proposée ici soit également

efficace pour ce type de système. Puisque les SOOC sont également basés sur le paradigme objet, un nombre restreint de modifications sont nécessaires. La structure générale demeure la même, et le processus de translation, du moins dans sa forme générale, ne comporte aucun changement majeur. Deux types de concurrence ont été discutés : la concurrence interne et la concurrence externe.

Concurrence interne

En ce qui concerne la concurrence interne, elle est déjà prise en considération. En effet, lorsqu'une classe présente un diagramme d'états-transitions dans lequel figure un état composite à régions orthogonales concurrentes, ce système présente alors les caractéristiques de la concurrence interne. Ce type de concurrence est donc considéré en utilisant des attributs d'états régionaux, introduits à la figure 4.9.

Concurrence externe

En ce qui concerne la concurrence externe, qui survient, par exemple, lorsque deux objets différents tentent tous les deux d'accéder en même temps à une ressource partagée, il s'agira d'en tenir compte lors de la création des règles de réécriture du module *COMMUNICATION*. Il faudra alors s'assurer que les objets concurrents du diagramme de communication soient tous les deux actifs en même temps, et que le comportement d'un de ces objets ne dépende pas de celui d'un autre. Il faudra aussi assurer la présence d'un mécanisme de coordination, tel que les sémaphores, au niveau de la ressource commune.

Afin de mieux illustrer tous ces éléments, le chapitre 5 étudiera trois exemples concrets. Le premier est un système classique dans lequel le processus de translation est appliqué directement. Le second montre des caractéristiques de concurrence interne, tandis que le troisième montre des caractéristiques de concurrence externe.

4.2.2 Validation par simulations

La particularité de la description formelle générée, sachant qu'elle a été développée à l'aide d'objets, de messages et de règles de réécriture, est qu'elle est exécutable. En effet, comme il a été étudié au chapitre 3, un programme Maude est à la fois un programme informatique et une théorie logique à partir de laquelle il est possible de faire plusieurs déductions.

Maude est un environnement très versatil en termes de simulation. En effet, puisqu'il est possible de définir un état initial personnalisé et d'exécuter cette configuration du système, il devient aisé de vérifier la description générée à l'aide de ce mécanisme. Ainsi, deux types de vérification par simulation peuvent être effectuées :

Vérification d'une règle de réécriture : Comme Maude permet à son utilisateur de définir un état initial personnalisé pour une simulation, il sera possible de vérifier le bon fonctionnement de chacune des règles de réécriture du module *COMMUNICATION*. Il sera même possible de le faire sans toutefois compromettre les autres règles ;

Vérification du système global : La seconde vérification à effectuer sera de simuler le système global pour vérifier que toutes les règles de réécriture mises en commun accomplissent bel et bien le comportement désiré.

Ces diverses vérifications sont illustrées plus en détail également dans le chapitre 5 à l'aide d'exemples concrets. Il est également possible de limiter le nombre de pas de réécriture que Maude effectue pour chacun des types de vérifications mentionnés précédemment. Cette possibilité pourra notamment être utilisée pour valider de façon plus complète le système en entier en allant vérifier une configuration intermédiaire. Encore ici, le lecteur est référé au chapitre 5 pour de plus amples détails à ce sujet.

4.2.3 Limites

Bien que ce processus soit intéressant, il a cependant des limites. Pour le moment, seuls les trois types de diagrammes mentionnés sont supportés par l'approche. Il pourrait être intéressant d'étendre le nombre de diagrammes considérés, mais cette étude dépasse le cadre de ce mémoire.

L'approche proposée considère également les structures les plus communes du diagramme de classes UML : l'héritage et l'agrégation.

En ce qui concerne le diagramme de communication UML, encore une fois les principales structures sont considérées. Il est important, par contre, de mentionner que les points de synchronisation entre messages sont directement supportés par l'utilisation de la structure *IsAccomplished* décrite un peu plus tôt. De plus, la concurrence externe entre objets est également prise en considération. En effet, puisqu'UML supporte directement les objets actifs, et qu'il ne s'agira que de tenir compte de cette particularité lors de l'échange des messages décrits par un diagramme de communication, la concurrence externe est prise en charge par l'approche proposée dans ce mémoire.

Enfin, en ce qui concerne les diagrammes d'états-transitions UML, seuls les éléments de base sont considérés. En effet, un diagramme d'états-transitions est une structure très riche en informations et regroupe un nombre important d'éléments. Pour le moment, seulement les transitions de type appel sont considérées, ainsi que les gardes sur les transitions. Les actions, les instructions d'entrée et les autres structures de ce genre de diagramme ne sont pas considérées pour le moment. Il serait par contre intéressant de vérifier de quelle façon ces éléments pourraient être intégrés dans l'approche, mais ceci dépasse une fois de plus le cadre de ce mémoire. Enfin, l'approche supporte la concurrence interne d'un objet. En effet, il est possible d'utiliser un attribut d'état pour chacune des régions orthogonales d'un état composite.

4.3 Vérification formelle de diagrammes UML

Jusqu'à maintenant, plusieurs modules ont été développés pour en arriver à décrire le comportement individuel et collectif de plusieurs objets collaborant en vue de l'accomplissement d'une tache donnée. Cette description formelle aura également été validée à l'aide de simulation. Comment pousser encore plus loin la vérification ?

Tel que le chapitre 2 l'a exploré, la technique de vérification de modèles a pour objectif de faire une analyse exhaustive de tous les chemins d'exécution possible d'un système et de

déterminer si une certaine propriété comportementale est satisfaite ou non. Cette propriété peut être de nature désirable ou non.

Ainsi, l'application de cette technique à la description formelle générée précédemment s'avérerait très intéressante pour valider les diagrammes UML développés dans le cadre du développement du système étudié. Ceci aurait l'avantage d'appliquer dès les premières étapes du processus de développement la technique de vérification de modèles et ainsi éviter la propagation d'erreurs subtiles, introduites au niveau des étapes d'analyse et de conception, au reste du processus de développement.

Le processus défini à la section précédente concernant la translation de diagrammes UML peut donc être étendu à quatre grandes étapes. La figure 4.12 présente visuellement ces quatres étapes. Les trois premières étapes sont les mêmes qui ont été discutées à la section 4.2 lors de la génération de la description formelle Maude. La quatrième étape, quant à elle, englobe le processus de vérification qui sera introduit dans ce qui suit. Ces quatre étapes peuvent être résumées visuellement par la figure 4.12 et par ce qui suit :

1. **Description d'un système :** Cette étape consiste en la description d'un système à l'aide des trois types de diagrammes UML utilisés : les diagrammes de classes, d'états-transitions et de communication. À ce stade, des propriétés comportementales du système doivent également être énoncées ;

2. **Validation inter-diagrammes :** Cette étape est la même que celle qui a été présentée à la section 4.2. Elle consiste en la validation des diagrammes entre eux ;

3. **Génération d'une description formelle Maude :** Cette étape est également la même que celle présentée à la section 4.2. Elle consiste en la génération de plusieurs modules Maude. Le module final, nommé *COMMUNICATION*, constitue la description finale et exécutable du système. Cette étape comprend également la validation par simulation des descriptions obtenues ;

4. **Validation de la description formelle Maude par vérification de modèles :** Cette étape est beaucoup plus longue que les autres. Elle consiste tout d'abord à reprendre les propriétés comportementales décrites à la première étape et de les transformer en propriété-

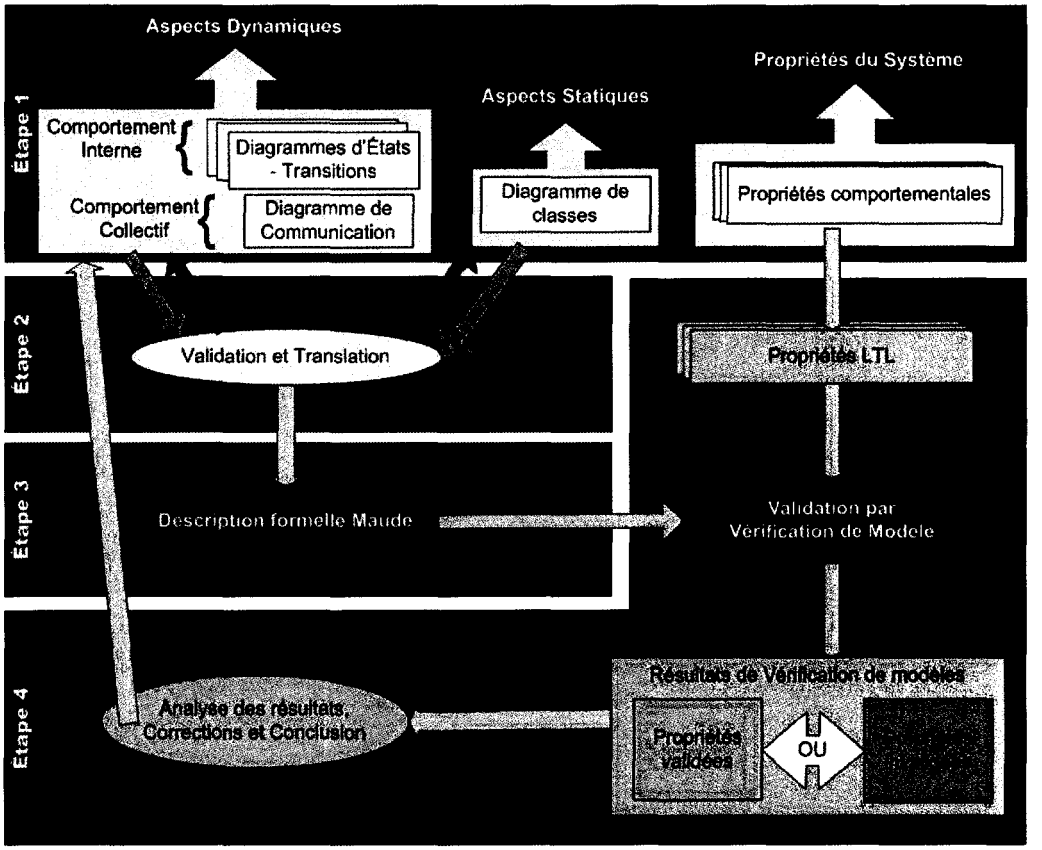


FIG. 4.12 – Approche combinée en quatre étapes

tés de LTL. Reprenant ces propriétés ainsi que la description formelle obtenue à l'étape 3, le processus continue par la vérification de modèles à l'aide du vérificateur intégré de l'environnement Maude. Enfin, avec les résultats obtenus, il faut tirer les conclusions appropriées et, si besoin est, revenir au modèle initial, lui apporter les correctifs nécessaires et reprendre le processus de vérification.

La première étape comporte un ajout : à la description du système à l'aide des trois types de diagrammes considérés, il faut aussi ajouter la description du comportement du système. L'étape d'analyse des besoins du processus de développement aura mis en lumière plusieurs éléments clés du comportement du système. L'analyste aura notamment fait ressortir, selon les attentes du client, ce que le système en développement aura à accomplir comme tâche. Il

aura mis en lumière les tâches critiques, ainsi que plusieurs éléments de sécurité importants. Il pourra même avoir mis en évidence certaines situations que le client considère comme inacceptables. Tous ces éléments sont directement liés au comportement du système. On les nomme alors propriétés comportementales.

Quant à elle, l'étape 4 constitue l'essentiel du processus de vérification de modèles. Elle se subdivise en quatre parties différentes. Elles sont :

1. Génération de Prédicats et d'États initiaux ;
2. Génération de propriétés LTL ;
3. Vérification de modèles ;
4. Analyse des résultats, Conclusion et, éventuellement, Correction.

Au chapitre 3, le procédé optimal d'utilisation du vérificateur de modèles de l'environnement Maude avait été illustré. Dans ce qui suit, un lien sera fait entre cette méthode et les étapes décrites précédemment. Chacune des étapes est reprise avec de plus amples informations.

1. Génération de Prédicats et d'États initiaux

Encore une fois, cette étape de travail se divise en deux parties distinctes : la génération d'une série de prédicats, et la génération d'états initiaux.

Au chapitre 3, une des étapes d'utilisation de l'environnement de vérification de modèles de Maude consistait en la définition de prédicats concernant le système en développement à l'aide de l'opérateur de satisfaction "$\models$". Pour ce faire, les développeurs de l'environnement Maude [22] suggèrent de développer un module *M-PREDS*. Comme *M* est dorénavant *COMMUNICATION*, il faut alors développer un nouveau module *COMMUNICATION-PREDICATES* qui contiendra les prédicats pertinents à l'étude comportementale du système.

Les prédicats définis sont simples. Il s'agit d'en définir un pour chacun des états de chacune des classes du système, tout en lui donnant un nom significatif et unique. Ces noms significatifs seront utilisés à nouveau plus loin dans le processus de vérification pour définir les propriétés comportementales sous la forme de formules de LTL.

Le type *Configuration* avait été utilisé pour décrire un système, donc ce dernier doit être défini comme le type de base pour la vérification. Ceci peut être accompli en insérant dans ce module la ligne *subsort Configuration < State* .

Dans plusieurs cas, il sera intéressant de définir l'état initial de vérification comme étant la séquence globale d'exécution du système. Par contre, pour certaines autres propriétés, il sera peut-être plus approprié de sélectionner un endroit autre pour débuter la vérification. Pour accomplir ceci, un nouveau module est défini. Ce dernier, nommé *COMMUNICATION-CHECK*, est l'équivalent du module *M-CHECK* suggéré par les auteurs du manuel de l'usager de Maude [22]. Il est également important de spécifier dans ce module quel type sera considéré comme le type de base pour l'analyse.

2. Génération de Propriétés LTL

Lors de cette étape, les propriétés comportementales décrites plus haut sont transformées en formules de LTL. Ceci est accompli grâce aux prédicats introduits à l'étape 1, ainsi qu'à l'aide des opérateurs de la LTL discutés au chapitre 3. Pour se rappeler ces opérateurs, voir le tableau 3.1 de la page 50.

Dans le manuel de l'usager de Maude [22], on définit directement les propriétés de LTL dans les appels à la fonction *modelCheck(initial_state, LTL_formulae)*, la fonction qui lance le vérificateur de modèles de l'environnement.

3. Vérification de modèles

Tous les éléments pertinents à la lancée de la vérification ont tous été définis précédemment, il s'agit de lancer l'exécution de la fonction *modelCheck* de Maude pour obtenir les résultats de l'analyse de vérification de chacune des propriétés de LTL définies.

4. Analyse des résultats, Conclusion et Correction

Après avoir obtenu les résultats, il faut également savoir les interpréter. Comment peut-on interpréter un résultat positif ou négatif à l'évaluation d'une propriété donnée ? La réponse

à cette question donne tout son sens à ce type d'analyse : en plus d'avoir obtenu un résultat positif ou un résultat de contre exemple à l'évaluation d'une formule de LTL, il faut savoir en tirer les bonnes conclusions.

Lorsque les résultats du processus de vérification auront été interprétés correctement et que les conclusions correspondantes auront été tirées, il restera encore l'étape de correction. En effet, si une situation problématique a été détectée, il faudra apporter quelques correctifs au modèle du système pour y remédier. Donc, il faudra fort probablement apporter des correctifs aux divers diagrammes utilisés et, par la suite, refaire l'analyse par vérification de modèles pour s'assurer que l'erreur détectée est bel et bien corrigée.

Il est également intéressant de noter que le processus utilisé pour la définition des propriétés à vérifier au sein du système est incrémental. D'abord, le comportement individuel des objets du système est vérifié, pour ensuite s'attarder au comportement collectif. Cette façon de faire sera adoptée tout au long des trois études de cas proposées au chapitre 5.

4.3.1 Discussion

Ceci complète donc l'intégration des deux grandes étapes majeures décrites aux sections 4.2 et 4.3. Le processus intégré de la figure 4.12 constitue un cadre formel par lequel trois types de diagrammes UML utilisés pour modéliser un système en développement sont validés à l'aide de deux techniques distinctes : les spécifications formelles et la vérification de modèles.

Le processus intégré décrit ici est innovateur dans le sens où il considère à la fois plusieurs aspects d'un même système. Tout d'abord, la structure statique du système est prise en considération par les diagrammes de classes UML. Les aspects dynamiques sont également considérés. En premier lieu, le comportement individuel des objets est représenté par des diagrammes d'états-transitions. Enfin, le comportement collectif est lui aussi intégré à l'approche. Les interactions dynamiques des objets du système sont décrits à l'aide de diagrammes de communication UML.

Chapitre 5

Études de Cas

Dans le chapitre 4, un cadre formel pour la translation et la vérification de diagrammes UML a l' aide de l'environnement Maude a été présenté. Ce processus, aux premiers abords, peut paraître abstrait et difficile à comprendre. Afin de mieux l'illustrer, trois études de cas sont présentées dans ce chapitre :

Le système d'Ascenseur : Exemple typique d'un système orienté-objet classique, typiquement séquentiel (Section 5.1) ;

Le système de Guichet Automatique : Exemple montrant un objet avec concurrence interne. Un objet avec concurrence interne n'est pas un système purement concurrentiel, mais fait plutôt référence au comportement individuel de cet objet décrit par un diagramme d'états montrant un super état a régions concurrentes (Section 5.2) ;

Le système de Producteur – Consommateur : Exemple d'un système concurrentiel où deux objets ou plus ont leur propre processus et peuvent tenter d'accéder à la même ressource en même temps. Ces objets sont alors dits "actifs" (Section 5.3).

Pour chacun de ces exemples, la même structure sera suivie afin d'illustrer le processus de translation et de vérification dans son ensemble. La structure de chacun des exemples aura donc l'allure suivante :

1. Présentation de l'exemple et des diagrammes UML qui lui sont associés ;

2. Application du processus de translation des diagrammes UML en une description formelle Maude, et validation de la description du système a l'aide de simulations ;
3. Application du processus de vérification formelle des diagrammes UML a l'aide de propriétés du système, de nature désirables et non désirables, énoncées a l'aide de la logique temporelle linéaire (LTL).

Le premier des trois exemples sera étudié dans le détail. Les deux autres, par contre, auront tous les détails pertinents sans toutefois aller dans les détails.

5.1 Le Système d'Ascenseur

Le premier exemple étudié est en réalité un système orienté-objet séquentiel des plus classiques. L'exemple utilisé, un système de gestion d'un ascenseur, a fait l'objet d'innombrables études. Ce cas a souvent été étudié comme exemple académique.

Tiré de [66], le système d'ascenseur a été simplifié pour les besoins de cet étude. Le problème sera divisé en trois parties distinctes. La première partie visera a introduire l'exemple et a expliquer les diagrammes UML qui lui sont rattachés. La seconde partie appliquera le processus de translation décrit dans le chapitre 4, tandis que la troisième partie appliquera le processus de vérification décrit au même chapitre.

5.1.1 Présentation

Le système étudié est conçu en vue de la gestion d'une cabine d'ascenseur. Ce système simplifié comprend quatres classes, tel qu'illustré le diagramme de classes de la figure 5.1. Ces quatre classes peuvent être décrites ainsi :

La classe *Door* : Cette classe a pour objectif de représenter et de gérer tout ce qui est en rapport avec la porte de la cabine d'ascenseur. Elle comporte un seul attribut, *StateD*, qui sera utilisé pour représenter l'état actuel de la porte de la cabine. Selon le diagramme d'états-transitions associé a cette classe (figure 5.2.(a)), un objet de cette classe pourra avoir deux états : *Opened* et *Closed*, respectivement pour quand la porte est ouverte et

fermée. La transition entre les deux états est déclenchée par la fonction qui commande l'ouverture de la porte *Open*, et la fonction qui commande la fermeture de la porte, *Close* ;

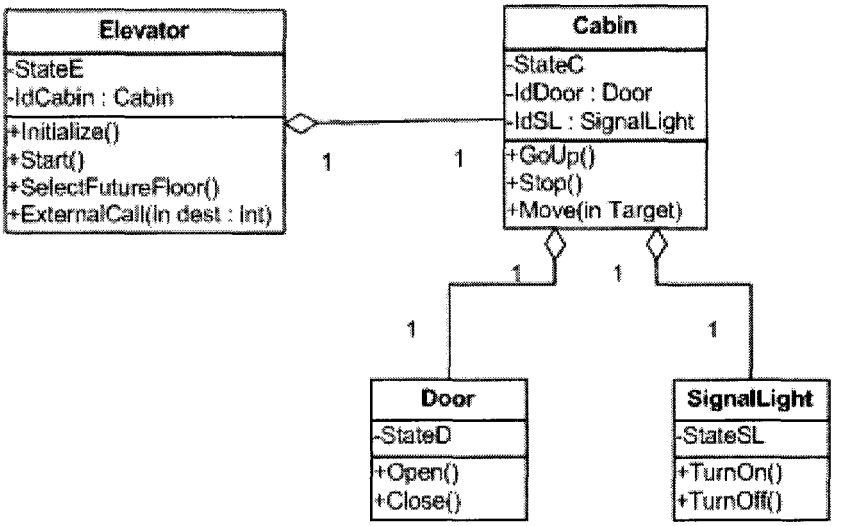


FIG. 5.1 – Diagramme de classes du système d'ascenseur

La classe *SignalLight* : Cette classe est utilisée pour représenter et gérer tout ce qui est en rapport avec les témoins lumineux normalement présents dans une cabine d'ascenseur pour indiquer à l'usager si cette dernière monte ou descend. Cette classe comporte également un seul attribut, *StateSL*, qui sera utilisé pour représenter l'état actuel d'un objet de cette classe. Deux valeurs d'états sont possibles (figure 5.2.(b)) : *On* et *Off*, respectivement pour représenter les cas lorsque le voyant est allumé et lorsqu'il est éteint. Les transitions entre les deux états sont déclenchées par les deux méthodes de la classe. La méthode *TurnOn* a pour effet d'allumer le voyant, tandis que *TurnOff* l'éteint ;

La classe *Cabin* : Cette classe est utilisée pour représenter tous les éléments d'une cabine d'ascenseur. On parle ici de la cabine physique. Cette cabine a la particularité d'être composée d'une porte et d'un voyant lumineux, tel que le démontrent les deux relations de composition du diagramme de classes de la figure 5.1. Cette classe a donc trois at-

tributs. Le premier, *StateC*, sera utilisé pour représenter l'état actuel d'un objet de cette classe. Selon le diagramme d'états-transitions associé à cette classe (figure 5.2.(c)), il existe deux états possibles : *Waiting* et *Moving*. Ces deux états représentent respectivement une cabine en attente et une cabine en mouvement. Les deux autres attributs de la classe sont des attributs d'identification des composants. Comme la cabine est composée d'une porte et d'un voyant, on devra déterminer quel objet porte et quel objet voyant sont associés à la cabine. Ces deux attributs seront utilisés pour stocker ces identificateurs d'objets. Enfin, la classe dispose de trois méthodes. La fonction *GoUp* déclenche le processus de démarrage de la cabine vers une destination sélectionnée par l'usager. La fonction *Stop* arrête le mouvement de la cabine, tandis que la méthode *Move* enclenche le mouvement ;

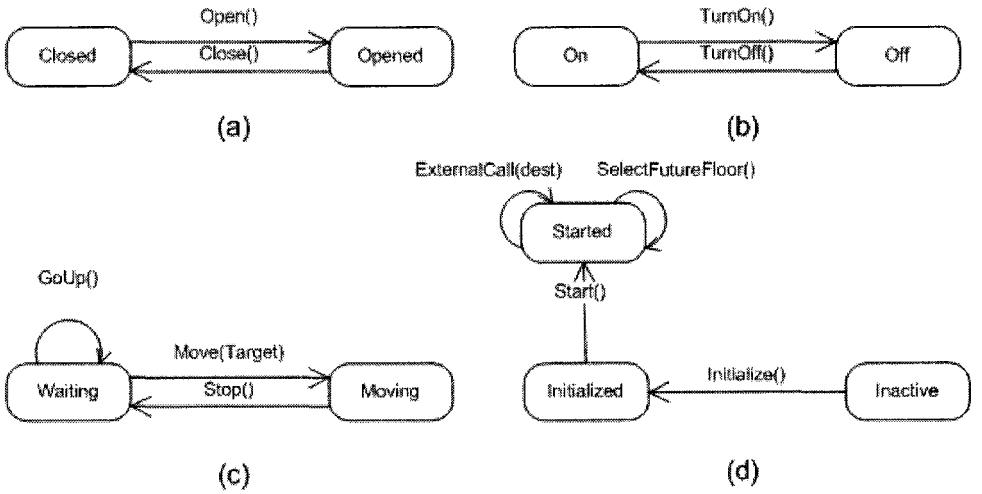


FIG. 5.2 – Diagramme d'états-transitions du système d'ascenseur, respectivement des classes (a) *Door*, (b) *SignalLight*, (c) *Cabin* et (d) *Elevator*

La classe *Elevator* : La dernière classe du système étudié est la classe principale du système, celle qui représente le système global. Cette classe est d'ailleurs composée d'une cabine, dont l'identificateur de cet objet sera stocké dans l'attribut *IdCabin*. L'autre attribut de la classe est *StateE* qui représentera l'état actuel d'un objet de cette classe. Les états possibles de ces objets sont donnés par le diagramme d'états-transitions as-

socié à cette classe, que l'on retrouve à la figure 5.2.(d). Ils sont : *Inactive*, *Initialized* et *Started*, respectivement pour lorsque le système d'ascenseur est inactif, lorsque le technicien de maintenance aura complété la procédure d'initialisation, et enfin lorsque le technicien de maintenance aura complété la procédure de démarrage.

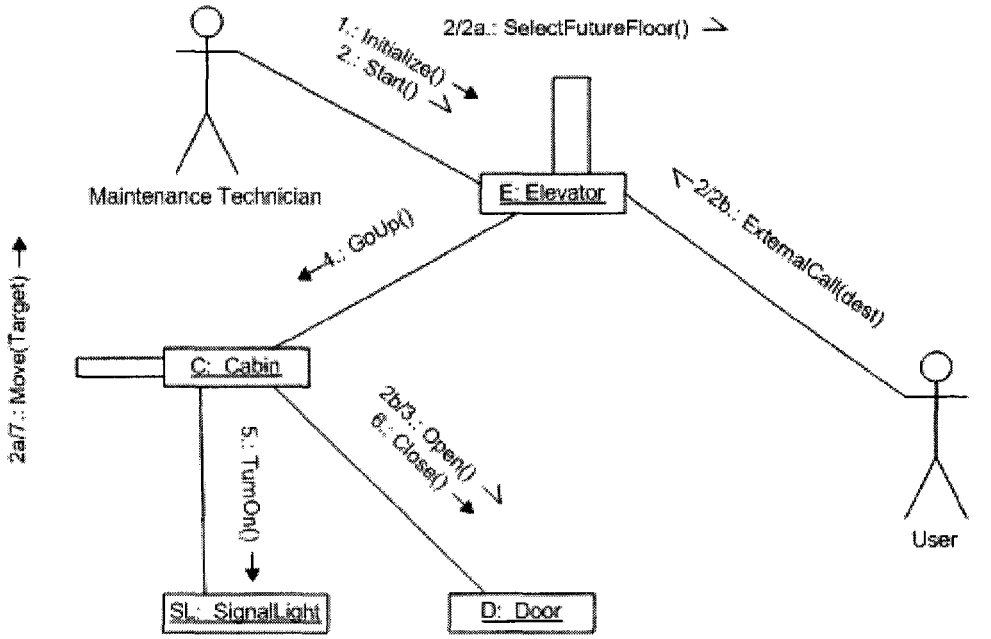


FIG. 5.3 – Diagramme de communication du système d'ascenseur

Une des fonctionalités du système étudié est décrite par le diagramme de communication de la figure 5.3. Ce diagramme décrit la séquence des messages envoyés entre objets pour accomplir la tâche de transporter un usager jusqu'à son étage de destination. Après l'initialisation et le démarrage du système d'ascenseur par un technicien de maintenance, un usager peut utiliser le système pour appeler l'ascenseur à l'étage où il se trouve, pour ensuite utiliser la cabine pour se rendre à un étage de son choix. Le diagramme de communication montre également à quel moment la porte doit s'ouvrir et se fermer, ainsi que le moment où le voyant lumineux doit être allumé et éteint.

5.1.2 Translation et Validation

La première partie de cette section applique le processus de translation décrit au chapitre 4, tandis que la seconde montre comment la description formelle ainsi générée peut être vérifiée à l'aide de simulations.

Translation

Dans la section qui suit, le processus de translation décrit au chapitre 4 sera appliqué à l'exemple du système d'ascenseur présenté plus tôt. On présume ici que les deux premières étapes de ce processus ont déjà été complétées adéquatement.

La première étape du processus de translation consiste en la définition des valeurs des états de chacune des classes, selon ce qui a été défini au sein des diagrammes d'états-transitions associés à chacune des classes du système. Comme la figure 5.2 montrait quatre diagrammes, un pour chacune des classes, on doit définir quatre modules. Ces modules porteront les noms suivants :

- *DOOR-STATEVALUES* ;
- *SIGNALLIGHT-STATEVALUES* ;
- *CABIN-STATEVALUES* ;
- *ELEVATOR-STATEVALUES.*

La figure 5.4 montre le module *ELEVATOR-STATEVALUES.* Les trois valeurs d'états possible de la clase *Elevator* y sont introduites et sont de type *ElevatorStateValues.*

```
fmod ELEVATOR-STATEVALUES is
  sort ElevatorStateValues .
  ops Inactive Initialized Started
              : -> ElevatorStateValues .
endfm
```

FIG. 5.4 – Le module *ELEVATOR-STATEVALUES*

Le diagramme de classes de la figure 5.1 montre que le système est composé de quatre classes distinctes : *Elevator*, *Cabin*, *SignalLight* et *Door.* Pour chacune de ces classes, un

module système est généré. Ce dernier contient, outre la déclaration de la classe et de ces attributs, la définition de chacune de ses méthodes. Les quatres modules ainsi générés sont :

- *DOOR* ;
- *SIGNALLIGHT* ;
- *CABIN* ;
- *ELEVATOR.*

De ces quatres modules, seul le module *CABIN* est donné ici (figure 5.5), de par sa nature intéressante. Voir les explications qui suivent à ce sujet.

```
mod CABIN is
  protecting DOOR SIGNALLIGHT CABIN-STATE-VALUES .
  *** CLass Decleration ****************************
  sort Cabin Target .
  subsort Cabin < Cid .
  op Cabin : -> Cabin .
  *** Attributes ***********************************
  op StateC :_ : CabinStateValues -> Attribute .
  op IdDoor :_ : Oid -> Attribute .
  op IdSL :_ : Oid -> Attribute .
  *** Methods **************************************
  subsort Target < Parameter .
  ops UP DOWN : -> Target .
  op GoUp :  ParameterList -> Void .
  op Stop :  ParameterList -> Void .
  op Move : Target  -> Void .
endm
```

FIG. 5.5 – Le module *CABIN*

Comme *Cabin* est en relation d'agrégation avec *Door* et *SignalLight*, deux attributs ont été définis pour stocker les identificateurs des objets de ces classes. Il faut aussi noter l'importation des modules définissant ces deux classes au début du module. Enfin, les trois méthodes de la classe sont définies à la fin du module. La méthode *Move* est un peu particulière : elle utilise un paramètre *Target* qui peut prendre deux valeurs spécifiques, *UP* et *DOWN.*

Attributs : Les attributs de la classe *Cabin* sont :

- *StateC* : sera utilisé pour représenter l'état actuel des objets de cette classe. Il est de type *CabinStateValues* ;
- *IdDoor* : la classe *Cabin* est composée d'un objet de type *Door*. Cet attribut aura pour fonction de stocker l'identificateur de l'objet *Door* qui est élément de la cabine d'ascenseur ;
- *IdSL* : la classe *Cabin* est composée d'un objet de type *SignalLight*. Cet attribut aura pour fonction de stocker l'identificateur de l'ojbet *SignalLight* qui est élément de la cabine d'ascenseur.

Méthodes : Les méthodes de la classe *Cabin* sont :

- *GoUp* : cette fonction, qui ne possède aucun paramètre, a pour objectif de démarrer la séquence qui aura pour effet, au bout du compte, de faire démarrer la cabine d'ascenseur. Elle déclenche successivement la fermeture de la porte, l'allumage le voyant lumineux, etc. ;
- *Stop* : cette fonction, également sans aucun paramètre, a pour effet de stopper la cabine d'ascenseur lorsque nécessaire ;
- *Move* : cette fonction a pour objectif de gérer le mouvement de la cabine. Elle possède un seul paramètre qui détermine la direction du déplacement : vers le haut (*UP*) ou vers le bas (*DOWN*).

Le prochain module généré est le module *IDENTIFICATION*. Ce module a pour objectif d'introduire un mécanisme adéquat d'identification des objets, tel qu'il est possible de le voir à la figure 5.6. Le module *CONFIGURATION* qu'importe le module *IDENTIFICATION* est en fait un module de l'environnement Maude. Quatre définitions de types y sont introduits : *Eoid*, *Coid*, *Doid* et *Soid*, qui décrivent respectivement les mécanismes d'identification des objets *E*, *C*, *D* et *SL* des classes *Elevator*, *Cabin*, *Door* et *SignalLight* respectivement.

Le module *COMMUNICATION* est le dernier module généré lors de l'application du processus de translation. Par contre, il en est aussi le module principal. Ce module apporte beaucoup de nouveaux éléments à la description formelle du système étudié. Il introduit le comportement individuel et collectif des objets en considérant à la fois les diagrammes

```
mod IDENTIFICATION is
  including CONFIGURATION .
  protecting STRING .
  sorts Eoid Coid Doid Soid Receiver .
  subsort Eoid Coid Doid Soid < Oid .
  subsort Receiver < Eoid Coid Doid Soid .
  subsort String < Eoid Coid Doid Soid .
endm
```

FIG. 5.6 – Le module *IDENTIFICATION*

d'états-transitions associés à chacune des classes présentées à la figure 5.2 et le diagramme de communication représentant l'approche utilisée pour accomplir une tâche donnée. Ce diagramme est présenté à la figure 5.3.

De façon plus précise, la tâche décrite par le diagramme de communication de la figure 5.3 consiste en la séquence de messages envoyés entre objets lorsqu'un usager du système utilise l'ascenseur pour atteindre un étage particulier.

Ce module importe tous les autres modules déjà définis jusqu'à maintenant. Les modules définissant les valeurs des états de chacune des classes sont omis sachant qu'ils sont déjà importés par les modules définissant les classes. La figure 5.7 montre une partie du module *COMMUNICATION*.

La partie montrée du module *COMMUNICATION* concerne plus particulièrement le comportement d'un objet de la classe *Elevator*. La première partie de la figure 5.7 montre les importations de modules ainsi que la définition générique des structures de message *ComingMsg* et des points de synchronisation *IsAccomplished*. Deux règles de réécriture sont montrées dans la figure 5.7. La première, identifiée par '*E1*', représente le comportement d'un objet *E* de la classe *Elevator* lorsqu'il reçoit un message *Initialize*. On remarque alors que l'objet *E* transite dans son état *Initialized* et qu'un message *IsAccomplished* est généré puisque la procédure de démarrage ne peut s'effectuer que lorsque l'initialisation de l'ascenseur est complétée. La deuxième règle, '*E2*', concerne quant à elle le comportement de *E* à la réception d'un message *Start*. On y remarque également que le message de synchronisation *IsAccomplished* est également consommé, signifiant que le point de synchronisation y sera

```
mod COMMUNICATION is
  protecting IDENTIFICATION .
  protecting CORRESPONDENCE .
  op ComingMsg : Event Receiver -> Msg .
  op IsAccomplished : Event Receiver -> Msg .
  var E : Eoid . var C : Coid . var D : Doid . var S : Soid .
*** Elevator's Behaviour ***********************************
  rl [E1]: ComingMsg(Initialize( EmptyParametersList ), E )
       < E : Elevator | StateE : Inactive, IdCabin : C >
  =>
       < E : Elevator | StateE : Initialized, IdCabin : C >
       IsAccomplished(Initialize( EmptyParametersList ), E)  .
  rl [E2]:  ComingMsg(Start( EmptyParametersList ), E )
       IsAccomplished(Initialize( EmptyParametersList ), E)
       < E : Elevator | StateE : Initialized, IdCabin : C >
  =>
       < E : Elevator | StateE : Started, IdCabin : C >
       IsAccomplished(Start( EmptyParametersList ), E)
       IsAccomplished(Start( EmptyParametersList ), E) .
...
endm
```

FIG. 5.7 – Le module *COMMUNICATION*

respecté. Il est également important de noter la génération de deux messages de synchronisation *IsAccomplished*. La raison pour laquelle deux messages de ce type sont générés est lié au fait que le déclenchement des messages *ExternalCall* et *SelectFutureFloor* dépend dans les deux cas de la complétion de la procédure de démarrage de l'ascenseur.

Validation de la Description Formelle

Jusqu'à maintenant, le processus de translation des diagrammes UML de classes, d'états-transitions et de communication du système classique de gestion d'un ascenseur a été complété. La description formelle ainsi obtenue peut maintenant faire l'objet d'une validation en utilisant l'environnement Maude, qui est très versatile en matière de simulations.

Maude est particulièrement intéressant à ce sujet car il permet de sélectionner une configuration initiale personnalisée pour une simulation, ce qui permettra de vérifier une partie du

système en ne compromettant pas ce qui a déjà été fait. Deux vérifications différentes seront effectuées ici : une première sur le comportement de l'objet *E* et des règles de réécriture qui lui sont propres, l'autre sur le comportement du système en entier.

Vérification du comportement de *E*. La première vérification qui sera tentée ici est décrite à l'aide de la configuration initiale donnée à la figure 5.8. On tente de vérifier que l'objet *E*, de la classe *Elevator*, se comporte correctement. En d'autres termes, on veut vérifier si les règles de réécriture qui ont été écrites pour cette classe fonctionnent correctement. Cette première simulation a pour objectif de vérifier que les deux premières règles, '*E1*' et '*E2*', sont bien formulées.

Dans la configuration initiale de la figure 5.8, on retrouve d'abord l'objet *E* dans son état initial, *Inactive*. Deux messages entrants y apparaissent également, *Initialize* et *Start*.

```
CommingMsg(Initialise( EmptyParamterList), E)
CommingMsg(Start( EmptyParamterList), E)
< E : Elevator | State : Inactive , IdCabin : C > .
```

FIG. 5.8 – Une configuration initiale pour vérifier le comportement de l'objet *E*

Les résultats de cette simulation sont donnés à la figure 5.9. L'objet *E* se retrouve alors dans son état démarré (*Started*), et deux messages de synchronisation *IsAccomplised* sont générés pour signaler la complétion du message *Start*. La raison pour laquelle il y a deux messages de ce type s'explique par le fait que l'exécution des messages *SelectFutureFloor* et *ExternalCall* dépend dans les deux de ce que la procédure de démarrage de l'ascenseur soit complétée.

```
IsAccomplished(Start(EmptyParametersList), E)
IsAccomplished(Start(EmptyParametersList), E)
< E : Elevator | StateE : Started, IdCabin : C >
```

FIG. 5.9 – Résultats de la configuration initiale de la figure 5.8

La seconde configuration initiale qui sera utilisée, qui se retrouve à la figure 5.10, est en fait une extension de la première. On y ajoute deux messages entrants, *SelectFutureFloor* et

ExternalCall. On viendra donc vérifier le bon fonctionnement des règles '*E3*' et '*E4*'.

```
CommingMsg(Initialise( EmptyParamterList), E)
CommingMsg(Start( EmptyParamterList), E)
< E : Elevator | State : Inactive , IdCabin : C > .
CommingMsg(ExternalCall( 4 ), E)
CommingMsg(SelectFutureFloor( EmptyParamterList), E)
```

FIG. 5.10 – Une seconde configuration initiale pour vérifier le comportement de l'objet *E*

Les résultats de cette simulation sont donnés à la figure 5.11. L'objet *E* se trouve alors dans son état démarré (*Started*) et un certain mouvement a été effectué comme en témoignent les deux messages *IsAccomplished*. En effet, l'ascenseur a été utilisé par un usager pour se rendre à sa destination. Dans le cas présent, l'usager a appelé la cabine depuis l'étage 4.

```
IsAccomplished(SelectFutureFloor(EmptyParametersList), E)
IsAccomplished(ExternalCall(4), E)
< E : Elevator | StateE : Started, IdCabin : C >
```

FIG. 5.11 – Résultats de la configuration initiale de la figure 5.10

Ces deux courtes simulations terminent la vérification du comportement d'un objet *E* de classe *Elevator*. Il aurait également été possible de procéder à la vérification de toutes les autres règles de réécriture par rapport à chacun des autres objets. Cette étape est cependant omise.

Vérification du système entier. Bien que le comportement individuel des objets ait été vérifié à l'aide de petites simulations, il reste maintenant à savoir si toutes les règles de réécriture mises en commun accomplissent le travail adéquatement : est-ce que le système global accomplit la tâche qui lui est assignée (plusieurs propriétés à vérifier) ?

Afin de vérifier la réponse à cette question, on procédera à une nouvelle simulation, mais cette fois-ci en prenant en considération le système en entier. Pour se faire, deux simulations différentes seront utilisées. Ces deux réécritures s'expliquent comme suit :

– La première réécriture, limitée à cinq pas, servira à visualiser l'état intermédiaire du

système. Il sera alors possible d'y visualiser l'évolution des états des objets, voir quels messages ont été consommés et ceux qui ne l'ont pas encore été ;

- La seconde réécriture, illimitée, permettra quant à elle de vérifier l'état final du système. On pourra alors vérifier si l'état final est un état cohérent du système ou non.

```
< E : Elevator | StateE : Inactive, IdCabin : C >
< C : Cabin | StateC : Waiting, IdDoor : D, IdSL : S >
< D : Door | StateD : Closed >
< S : SignalLight | StateSL : Off >
ComingMsg(Initialize( EmptyParametersList ), E )
ComingMsg(Start( EmptyParametersList ), E )
ComingMsg(ExternalCall( 4 ), E )
ComingMsg(SelectFutureFloor( EmptyParametersList ), E )
ComingMsg(GoUp( EmptyParametersList ), C )
ComingMsg(Move(UP), C )
ComingMsg(Open( EmptyParametersList ), D )
ComingMsg(Close( EmptyParametersList ), D )
ComingMsg(TurnOn( EmptyParametersList ), S ) .
```

FIG. 5.12 – Configuration initiale pour la vérification du système global

Les deux réécritures proposées utilisent la même configuration initiale (présentée à la figure 5.12). Cette configuration comporte les éléments suivants :

- Quatre objets, *E*, *C*, *D* et *S*, tous dans leurs états initiaux ;
- Une série de messages entrants destinés aux divers objets composant le système. Cette série de messages est en fait la séquence montrée par le diagramme de communication de la figure 5.3.

Les résultats de ces deux simulations sont donnés par la saisie d'écran de la figure 5.13. On y voit apparaître les résultats des deux réécritures. Le résultat de la réécriture limitée à cinq pas permet de visualiser les états intermédiaires des objets, ainsi que les messages consommés et les messages qui ne l'ont pas encore été. Enfin, le résultat de la réécriture illimitée permet de visualiser l'état final du système. On peut alors constater que le système termine son exécution dans un état cohérent pour un système d'ascenseur de ce genre.

FIG. 5.13 – Résultats de la simulation du système d'ascenseur

5.1.3 Vérification Formelle

Au chapitre 4, une méthode avait été étudiée afin d'appliquer les techniques de vérification de modèles à la description formelle générée. Ce processus de vérification formelle sera appliqué ici sur le système de gestion d'un ascenseur.

Les deux sections qui suivent introduiront respectivement une série de propriétés du comportement individuel et collectif des objets du système. Par la suite, une section décrira plus en détail le processus adopté pour procéder à la vérification, et la dernière partie verra à analyser les résultats obtenus.

Propriétés du comportement individuel

Dans ce qui suit, sept propriétés sont proposées pour procéder à la vérification des objets du système. De ces propriétés, les Propriétés 1 et 2 tentent de vérifier le comportement de l'objet *E* de classe *Elevator*. La Propriété 3, quant à elle, vérifie le comportement de l'objet *C* de classe *Cabin*. Les Propriétés 4 et 5 font de même pour l'objet *D* de classe *Door*, tandis

que les Propriétés 6 et 7 le font pour l'objet *S* de classe *SignalLight*.

Propriété 1 :

- Formule LTL :

```
Elevator-In-Inactive-State("E") ->
    (O Elevator-In-Initialized-State("E"))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialE1* (figure 5.15), cette propriété exprime le fait que si l'objet "*E*" se trouve dans son état *Inactive*, la propriété *Elevator-In-Initialized-State*, exprimant que "*E*" soit dans son état *Initialized*, est vraie dans le prochain état (noter que *O* est l'opérateur temporel *Next* et que -> est l'opérateur temporel *Implies*).

Propriété 2 :

- Formule LTL :

```
Elevator-In-Inactive-State( "E" ) ->
    <>( Elevator-In-Started-State( "E" ))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialE2* (figure 5.15), cette propriété tente de vérifier que si l'objet *E* est dans son état *Inactive*, il sera éventuellement dans son état *Started*. Noter que <> est l'opérateur temporel *Eventually*.

Propriété 3 :

- Formule LTL :

```
Cabin-In-Moving-State( "C" ) ->
     (O Cabin-In-Waiting-State( "C" ))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialC1* (figure 5.15), cette propriété exprime le fait que si l'objet *C* se trouve dans son état *Moving*, la propriété *Cabin-In-Waiting-State*, exprimant le fait que la cabine est dans son état *Waiting*, est vraie dans le prochain état.

Propriété 4 :

- Formule LTL :

```
SignalLight-In-On-State( "S" ) ->
    (O SignalLight-In-Off-State( "S" ))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialSL1* (figure 5.15), cette propriété vérifie que si l'objet *S* est dans son état *On*, il sera alors dans son état *Off* dans le prochain état.

Propriété 5 :

- Formule LTL :

```
SignalLight-In-Off-State( "S" ) ->
    (O SignalLight-In-On-State( "S" ))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialSL2* (figure 5.15), cette propriété vérifie que si l'objet *S* est dans son état *Off*, il sera alors dans son état *On* dans le prochain état. Cette propriété est la contrepartie de la propriété précédente.

Propriété 6 :

- Formule LTL :

```
Door-In-Opened-State( "D" ) ->
    (O Door-In-Closed-State( "D" ))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialD1* (figure 5.15), cette propriété exprime le fait que si l'objet *D* est dans son état *Closed*, il sera dans son état *Opened* dans le prochain état.

Propriété 7 :

- Formule LTL :

```
Door-In-Closed-State( "D" ) ->
    (O Door-In-Opened-State( "D" ))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialD2* (figure 5.15), cette propriété exprime le fait que si l'objet *D* est dans son état *Opened*, il sera dans son état *Closed* dans le prochain état. Cette propriété est la contrepartie de la propriété précédente.

Propriétés du comportement collectif

À la section précédente, des propriétés ont été énoncées en vue de vérifier le comportement individuel des objets *E*, *C*, *D* et *S*. Les sept propriétés suivantes auront plutôt pour objectif de vérifier que ces mêmes objets se comportent correctement lorsqu'ils collaborent.

Propriété 8 :

- Formule LTL :

```
[] (Cabin-In-Moving-State( "C" ) ->
    Elevator-In-Started-State( "E" ))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialCS3* (figure 5.15), cette propriété tente de vérifier s'il est toujours vrai que l'objet *C* est dans son état *Moving* alors que l'objet *E* est dans son état *Started*. Ceci est une propriété désirable puisque le système ne devrait permettre à une cabine d'être en mouvement que lorsque le système a été démarré. Noter que "[]" est l'opérateur temporel *Henceforth*.

Propriété 9 :

- Formule LTL :

```
<> (Cabin-In-Moving-State( "C" ) /\
    Elevator-In-Inactive-State( "E" ))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialCS3* (figure 5.15), cette propriété a pour objectif de vérifier si, éventuellement, le système permet à l'objet *C* d'être dans son état *Moving* au même moment où l'objet *E* est dans son état *Inactive*. Cette propriété est donc de nature non désirable.

Propriété 10 :

- Formule LTL :

```
[] (SignalLight-In-On-State( "S" ) ->
    Door-In-Closed-State( "D" ))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialCS3* (figure 5.15), cette propriété vérifie qu'il est toujours vrai que l'objet *S* est

dans son état *On* alors que l'objet *D* est dans sont état *Closed.* Puisque le voyant lumineux a pour objectif de signaler le mouvement de la cabine, il serait correct de retrouver une telle situation.

Propriété 11 :

- Formule LTL :

```
[] (Cabin-In-Moving-State( "C" ) ->
    Door-In-Closed-State( "D" ))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialCS3* (figure 5.15), cette propriété tente de vérifier s'il est toujours vrai que lorsque l'objet *C* est dans son état *Moving*, l'objet *D* est dans son étant *Closed.* Cette situation est particulièrement désirable pour la sécurité des usagers : il serait très dangereux que le système permette à une cabine de se déplacer avec la porte ouverte.

Propriété 12 :

- Formule LTL :

```
<> (Cabin-In-Moving-State( "C" ) /\
    Door-In-Opened-State( "D" ))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialCS3* (figure 5.15), cette propriété tente de vérifier s'il sera éventuellement vrai que l'objet *C* soit dans son état *Moving* au même moment où l'objet *D* est dans son état *Opened.* Comme la sécurité des usagers est primordiale, une seconde propriété a été définie pour s'assurer que le système ascenseur ne permette pas à la cabine d'être en mouvement avec la porte ouverte. Ici, on veut savoir si ce sera éventuellement possible. Cette situation est donc non désirable de nature.

Propriété 13 :

- Formule LTL :

```
[] (Coherent-System-State("E", ("C", ("D", "S"))))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialCS1* (figure 5.15), cette propriété tente de vérifier si le système est toujours dans

TAB. 5.1 – Les états cohérents du système d'ascenseur

Classe ELEVATOR	Classe CABIN	Classe DOOR	Classe SIGNALLIGHT
Inactive	Waiting	Opened	Off
Inactive	Waiting	Closed	Off
Initialized	Waiting	Opened	Off
Initialized	Waiting	Closed	Off
Started	Moving	Closed	On
Started	Waiting	Closed	Off
Started	Waiting	Opened	Off

un état cohérent. Le tableau 5.1 montre tous les états cohérents du système. Cette propriété est donc vraie pour un des cas énoncés dans le tableau.

Propriété 14 :

- Formule LTL :

```
[] (Coherent-System-State("E", ("C", ("D", "S"))))
```

- Description : En démarrant la vérification à partir de la configuration initiale *initialCS2* (figure 5.15), cette propriété tente de vérifier si le système est toujours dans un état cohérent. Le tableau 5.1 montre tous les états cohérents du système. Cette propriété est donc vraie pour un des cas énoncés dans le tableau. Cette propriété est analogue à la propriété précédente. La différence entre les deux réside dans la configuration initiale par laquelle a débuté la vérification. Dans le cas présent, *initialCS2* n'est pas un des cas énoncés dans le tableau, ce qui fait de cette propriété un élément non désirable.

Afin d'énoncer les 14 propriétés précédentes, plusieurs prédicats ont été utilisés (par exemple, *Cabin-In-Moving-State("C")* en est un). La figure 5.14 présente une partie du module *COMMUNICATION-PREDICATES* dans lequel sont définis tous ces prédicats. Les prédicats montrés dans la figure sont limités aux trois relatifs à l'objet *E* de la classe *Elevator*. On remarque alors qu'un prédicat est défini par valeur d'état possible (voir le diagramme d'états-transitions de la classe *Elevator* à la figure 5.2.(d)). Les lignes 1, 2 et 3 sont les prédicats définis pour les états *Inactive*, *Initialized* et *Started* respectivement.

```
mod COMMUNICATION-PREDICATES is
  protecting COMMUNICATION .
  including SATISFACTION .
  subsort Configuration < State .
  ...
  *** Operators ********************************************
  op Coherent-System-State : Configuration : OidList -> Prop .
  ops Elevator-In-Inactive-State
      Elevator-In-Initialized-State
      Elevator-In-Started-State : Oid -> Prop .
  ops Cabin-In-Waiting-State
      Cabin-In-Moving-State : Oid -> Prop .
  ops SignalLight-In-On-State
      SignalLight-In-Off-State : Oid -> Prop .
  ops Door-In-Closed-State
      Door-In-Opened-State : Oid -> Prop .
  *** Variables ********************************************
  vars SE SE1 : ElevatorStateValues .
  var SC : CabinStateValues .    var SD : DoorStateValues .
  var SS : SignalLightStateValues .  var EV : Event .
  vars E C D S O : Oid .  var conf : Configuration .
*** Elevator's Internal Behaviour ****************************
eq  < E : Elevator | StateE : Inactive, IdCabin : C > conf
    |= Elevator-In-Inactive-State( "E" ) = true .        ***[1]
eq  < E : Elevator | StateE : Initialized, IdCabin : C > conf
    |= Elevator-In-Initialized-State( "E" ) = true .     ***[2]
eq  < E : Elevator | StateE : Started, IdCabin : C > conf
    |= Elevator-In-Started-State( "E" ) = true .         ***[3]
************************************************************
...
*** Confirm System is always in a Coherent State *************
ceq  < E : Elevator | StateE : SE, IdCabin : C >
     < C : Cabin | StateC : SC, IdDoor : D, IdSL : S >
     < D : Door | StateD : SD >
     < S : SignalLight | StateSL : SS > conf
  |= Coherent-System-State( "E", ("C", ("D", "S")))  = true
  if CorrespondingState( SE, (SC, (SD, SS))) == true .   ***[4]
************************************************************
endm
```

FIG. 5.14 – Une partie du module *COMMUNICATION-PREDICATES*

Le prédicat *Coherent-System-State("E", ("C", ("D", "S")))* (ligne 4) est également présenté étant donné sa nature différente des autres. En premier lieu, ce prédicat est conditionnel au résultat obtenu à l'évaluation de la fonction *CorrespondingStates*. Cette fonction retourne un résultat positif (*True*) dans les cas énumérés dans le tableau 5.1.

Vérification des propriétés

Lorsque les 14 propriétés ont été énoncées, il a été question de plusieurs états initiaux par lesquels débutent les vérifications de chacune de ces propriétés. En plus d'introduire ces configurations initiales, le module *COMMUNICATION-CHECK*, montré à la figure 5.15, établit le lien final entre le système étudié et le vérificateur de modèles de Maude. Étant donné le grand nombre de configurations initiales utilisées pour la vérification des 14 propriétés, seulement quelques-unes sont montrées ici. Les lignes 1 et 2 montrent les états initiaux utilisés pour la vérification du comportement individuel des objets de la classe *Elevator*. Les lignes 3 et 4, quant à elles, réfèrent à la vérification du comportement collectif des objets. La ligne 4 est la même configuration initiale qui avait été utilisée lors de la simulation du système global. La ligne 3 fait plutôt référence à la Propriétés 13 qui tente de vérifier l'état global du système.

Avec le module *COMMUNICATION-CHECK*, tous les éléments nécessaires à la vérification formelle du système d'ascenseur sont maintenant réunis. Il ne reste plus qu'à lancer la vérification à l'aide du vérificateur de modèle de Maude. Pour ce faire, on doit faire appel à la fonction *modelCheck* du module *MODEL-CHECKER* de Maude. Cette fonction a comme particularité de retourner un résultat positif (*True*) lorsqu'une propriété est vérifiée, ou un résultat de contre exemple lorsqu'une propriété ne l'est pas. Le contre exemple sera particulièrement utile pour déterminer le chemin d'exécution emprunté par Maude avant d'atteindre un état d'erreur. Ce chemin apportera très fort probablement un élément de solution lors de la correction de l'erreur trouvée. La fonction *modelCheck* de Maude nécessite l'utilisation de deux paramètres :

State : Ce paramètre est en réalité l'état initial à partir duquel la vérification doit débuter ;

```
mod COMMUNICATION-CHECK is
  including COMMUNICATION-PREDICATES .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER .
  *** Operators ********************************************
  ops initialE1 initialE2 initialC1 initialSL1
      initialSL2 initialD1 initialD2 initialCS1
      initialCS2 initialCS3 : -> Configuration .
  *** Elevator's initial states ****************************
  eq initialE1 =                                          ***[1]
      ComingMsg(Initialize( EmptyParametersList ), "E" )
      < "E" : Elevator | StateE : Inactive, IdCabin : "C" > .
  eq initialE2 =                                          ***[2]
      ComingMsg(Initialize( EmptyParametersList ) , "E" )
      ComingMsg(Start ( EmptyParametersList ), "E")
      < "E" : Elevator | StateE : Inactive, IdCabin : "C" > .
...
  *** Collective behavior's initial states *****************
  eq initialCS1 =                                         ***[3]
  < "E" : Elevator | StateE : Inactive, IdCabin : "C" >
  < "C" : Cabin | StateC : Waiting, IdDoor : "D", IdSL : "S" >
  < "D" : Door | StateD : Closed >
  < "S" : SignalLight | StateSL : Off > .
  eq initialCS3 =                                         ***[4]
  < "E" : Elevator | StateE : Inactive, IdCabin : "C" >
  < "C" : Cabin | StateC : Waiting, IdDoor : "D", IdSL : "S" >
  < "D" : Door | StateD : Closed >
  < "S" : SignalLight | StateSL : Off >
  ComingMsg(Initialize( EmptyParametersList ), "E" )
  ComingMsg(Start( EmptyParametersList ), "E" )
  ComingMsg(ExternalCall( 4 ), "E" )
  ComingMsg(SelectFutureFloor( EmptyParametersList ), "E" )
  ComingMsg(GoUp( EmptyParametersList ), "C" )
  ComingMsg(Move(UP), "C" )
  ComingMsg(Open( EmptyParametersList ), "D" )
  ComingMsg(Close( EmptyParametersList ),"D" )
  ComingMsg(TurnOn( EmptyParametersList ), "S" ) .
endm
```

FIG. 5.15 – Une partie du module *COMMUNICATION-CHECK*

Formulae : Ce paramètre est en réalité la formule de LTL qui doit être vérifiée au sein du système.

Comme 14 propriétés ont été définies, il faut lancer 14 appels à la fonction *modelCheck* pour obtenir tous les résultats de l'évaluation des propriétés. La figure 5.16 montre quelques-uns des appels à cette fonction.

```
red modelCheck(initialE1, ( Elevator-In-Inactive-State( "E" )
      -> (O Elevator-In-Initialized-State( "E" )))) .
red modelCheck(initialE2, Elevator-In-Inactive-State( "E" ) ->
      (<> Elevator-In-Started-State( "E" ))) .
...
red modelCheck(initialCS3, <> (Cabin-In-Moving-State( "C" ) /\
                              Door-In-Opened-State( "D" ))) .
red modelCheck(initialCS1,
        [] (Coherent-System-State("E", ("C", ("D", "S"))))) .
...
```

FIG. 5.16 – Quelques appels à la fonction *modelCheck*

Une partie des résultats de l'évaluation de ces 14 propriétés sont montrés dans la capture d'écran de la figure 5.17. De plus, un résumé des résultats est donné par le tableau 5.2, où l'on retrouve, pour chacune des propriétés, un résultat positif ou un résultat de contre exemple. Sur les 14 propriétés, 11 ont obtenu un résultat positif, et trois ont obtenu un résultat de contre exemple. La section suivante verra à interpréter ces résultats.

Analyse des résultats et Conclusion

Tel que mentionné au chapitre 4, il ne suffit pas d'obtenir le résultat de l'évaluation des propriétés définies. Il faut aussi savoir interpréter ces résultats, les mettre en contexte, et en tirer les bonnes conclusions pour être en mesure, lorsque cela est nécessaire, d'apporter les bons correctifs.

Résultats du comportement individuel : Tel que vu précédemment, les Propriétés 1 à 7 faisaient référence au comportement individuel des objets impliqués. L'analyse de ce com-

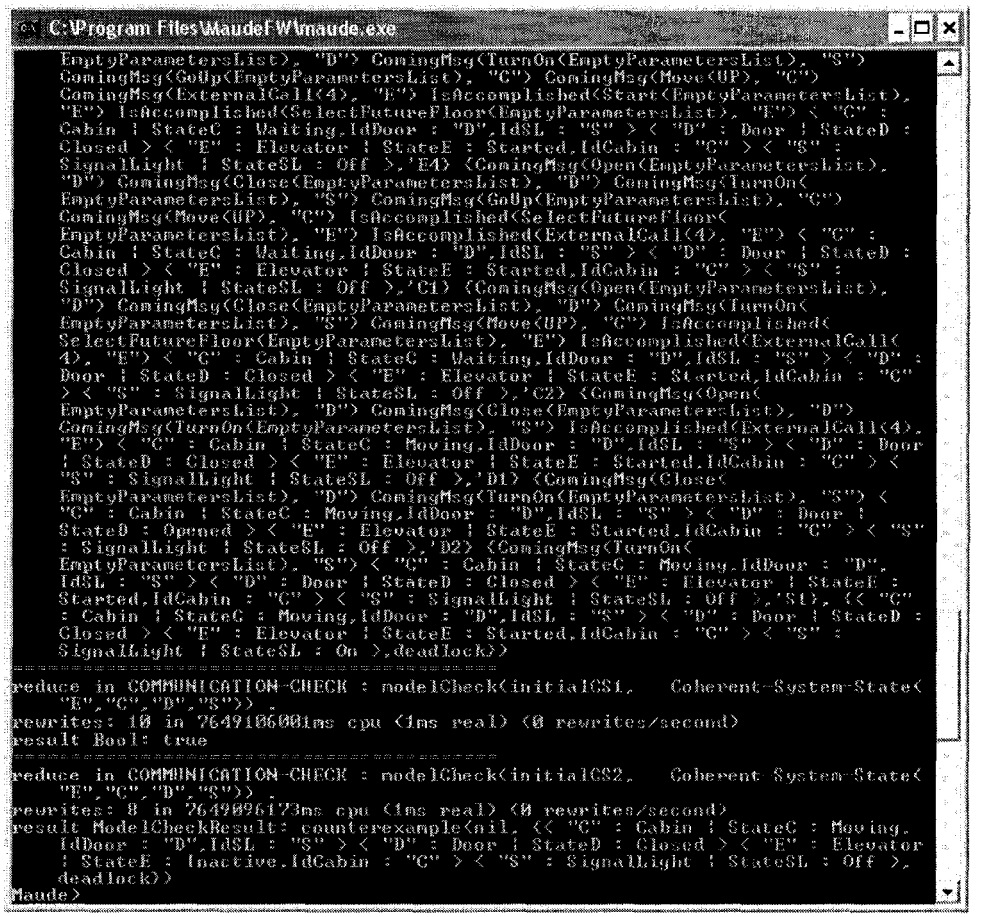


FIG. 5.17 – Partie des résultats des appels à la fonction *modelCheck*

portement a été l'objet de la première série de vérifications. Ces différentes propriétés ont été vérifiées.

Des Propriétés 1 et 2, on conclue que le comportement des objets de la classe *Elevator* est adéquat. En d'autres termes, l'état suivant *Inactive* est *Initialized*, et éventuellement, l'objet atteint son état *Started*. Ceci répond parfaitement à ce qui avait été spécifié par le diagramme d'états-transitions de la classe *Elevator* que l'on retrouve à la figure 5.2.(d).

De la même façon, on peut conclure que les objets de la classe *Cabin* se comportent également correctement. Un objet de cette classe atteint son état *Moving*.

Les objets de la classe *SignalLight* atteignent également leurs deux états possibles, *On* et

TAB. 5.2 – Résultats de l'évaluation des 14 propriétés du problème de l'Ascenseur

Propriété	**Résultat**	**Propriété**	**Résultat**
1	Vrai	8	Vrai
2	Vrai	9	Contre-exemple
3	Vrai	10	Vrai
4	Vrai	11	Vrai
5	Vrai	12	Contre-exemple
6	Vrai	13	Vrai
7	Vrai	14	Contre-exemple

Off. Les Propriétés 4 et 5 s'en assuraient. Il est bon de noter que la Propriété 4 vérifiait qu'un objet *S* s'allumait, tandis que la Propriété 5 vérifiait l'inverse.

De façon similaire à la classe *SignalLight*, les objets de la classe *Door* ont aussi un comportement individuel correct. Les Propriétés 6 et 7 étaient construites pour vérifier cela. La Propriété 6 vérifiait que la porte se ferme (transige vers son état *Closed*), et la Propriété 7 qu'elle s'ouvre (transige vers son état *Opened*).

Résultats du comportement collectif : Les Propriétés 8 à 14 avaient plutôt comme objectif la vérification du comportement collectif des objets *E*, *C*, *D* et *S*, respectivement des classes *Elevator*, *Cabin*, *Door* et *SignalLight*. De ces sept propriétés, les propriétés 8, 10, 11 et 13 ont obtenu un résultat positif (elles ont été vérifiées au sein du système). Les Propriétés 9, 12 et 14 ont, quant à elles, obtenu un résultat de contre-exemple.

Rappelons que la Propriété 8 avait pour objectif de vérifier si le fait que l'objet *C* était dans son état *Moving* impliquait toujours le fait que l'objet *E* soit dans son état *Started*. On tentait ici de s'assurer si la cabine est en mouvement, le système d'ascenseur est toujours bel et bien démarré.

La Propriété 9, quant à elle, voulait vérifier s'il était éventuellement possible d'avoir à la fois l'objet *C* dans son état *Moving* alors que l'objet *E* est dans son état *Inactive*. Cette

propriété était de nature non désirable au sein du système étudié. Comme le résultat obtenu est un contre-exemple, cela signifie que la propriété en question n'a pas été vérifiée et, par le fait même, cette situation est impossible au sein du système d'ascenseur.

La Propriété 10, qui a été vérifiée, s'assurait qu'un objet *S* dans son état *On* impliquait que la porte de l'ascenseur soit dans son état *Closed*. Comme la fonction du voyant lumineux du système d'ascenseur est de signaler que la cabine est en mouvement, il est normal que le voyant s'allume lorsque la porte est fermée.

Les Propriétés 11 et 12 ont pour objectif la vérification de la porte de l'ascenseur en conjonction avec le mouvement de la cabine. Il serait particulièrement dangereux que le système permette à la cabine de bouger alors que la porte est ouverte. La Propriété 11 vérifie qu'il est toujours vrai que l'objet *C* soit dans son état *Moving* au même moment où l'objet *D* soit dans son état *Closed*. Cette propriété a été vérifiée. De son côté, la Propriété 12 vérifie s'il est éventuellement vrai que l'objet *C* soit dans son état *Moving* alors que l'objet *D* est dans son état *Opened*. La nature de cette propriété est non désirable. Comme le résultat obtenu est un contre-exemple, la propriété n'a pas été vérifiée.

Les Propriétés 13 et 14 tentaient quant à elles de vérifier l'état global du système. À titre de rappel, les états correspondants à un état global du système cohérent sont donnés dans le tableau 5.1. La propriété 13, dont la vérification a été démarrée à partir d'un état initial dit cohérent, a obtenu un résultat positif. Quant à elle, la Propriété 14 n'a pas été vérifiée puisque son état initial était non cohérent.

5.2 Le Système de Guichet Automatique

L'étude de cas présentée ici est un système de gestion d'un guichet automatique simple. Tiré de [78], cet exemple a été adapté et simplifié pour ce mémoire. Le système étudié est un système simple illustrant la communication qui existe entre un guichet automatique et la banque. Le problème étudié ici consiste en une opération de retrait effectuée par un usager. La particularité de cet exemple est la présence d'un objet montrant un comportement interne présentant des caractéristiques de concurrence.

5.2.1 Présentation

Le système étudié consiste en un système de gestion d'un guichet automatique simple. Le système comporte deux classes : la classe *ATM* et la classe *Bank*, tel que le montre le diagramme de classes UML de la figure 5.18.

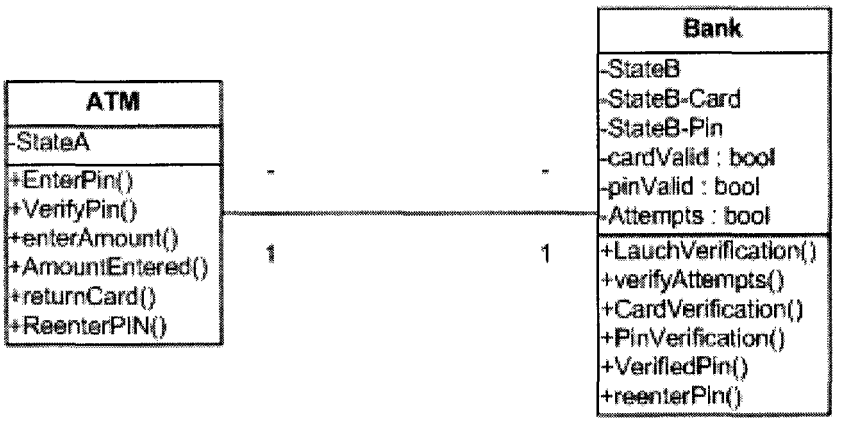


FIG. 5.18 – Diagramme de classes du système de guichet automatique

La classe *ATM* : Cette classe représente le guichet automatique qu'un usager utilise pour accomplir diverses opérations sur son compte de banque personnel. Pour s'identifier, un usager doit disposer de deux éléments :

- D'une carte bancaire valide émise par la banque ;
- De son numéro d'identification personnel (NIP).

Sans ces deux éléments, l'usager ne pourra utiliser les services offerts par le guichet automatique. Le but de la communication entre un objet de la classe *ATM* et un objet de la classe *Bank* a pour objectif d'identifier l'usager par sa carte bancaire valide et son NIP.

La classe *ATM* ne dispose que d'un seul attribut, *StateA*, qui représente l'état actuel d'un objet de cette classe. Par contre, elle dispose de plusieurs méthodes. Ce sont celles qui sont utilisées dans une séquence normale pour qu'un usager puisse effectuer un retrait. La première méthode, *EnterPin*, demande à l'usager d'entrer son NIP au clavier. La méthode *VerifyPin* lance quant à elle la vérification de la validité de la carte bancaire et du NIP de l'usager auprès de la banque (un objet de la classe *Bank*).

La méthode *ReenterPIN* a pour objectif de couvrir l'éventualité où l'usager aurait mal introduit son NIP au clavier, tel que le montre le diagramme d'états-transitions associé à cette classe. Ce diagramme peut être retrouvé à la figure 5.19.(a). Enfin, les méthodes *enterAmount*, *AmountEntered* et *returnCard* remettent l'argent à l'usager ;

La classe *Bank* : Cette classe représente le système central de la banque de l'usager. La tâche dévolue à un objet de cette classe est de vérifier la validité d'une carte bancaire et du NIP d'un usager. La classe *Bank* comporte six attributs :

- *StateB* sera utilisé pour représenter l'état actuel d'un objet de la classe *Bank* ;
- *Attempts* comptabilise le nombre de tentatives faite par l'usager pour entrer le bon NIP. Comme le montre le diagramme d'états-transitions de la classe *Bank* (que l'on trouve à la figure 5.19.(b)), le nombre de tentatives autorisé est de trois ;
- Les attributs *StateB-Pin* et *StateB-Card* sont des attributs d'états régionaux. En effet, le diagramme d'états-transitions de la figure 5.19.(b) montre que l'état *Verifying* est un état composite avec deux régions orthogonales concurrentes ;
- Les attributs *cardValid* et *pinValid*, tous les deux booléens, sont utilisés pour signifier si la carte bancaire et le NIP sont valides. Ce mécanisme est utilisé pour simplifier l'exemple.

Tel que mentionné précédemment, la classe *Bank* a un état composite avec deux régions orthogonales. Ces deux régions orthogonales ont pour tâche respective de vérifier la

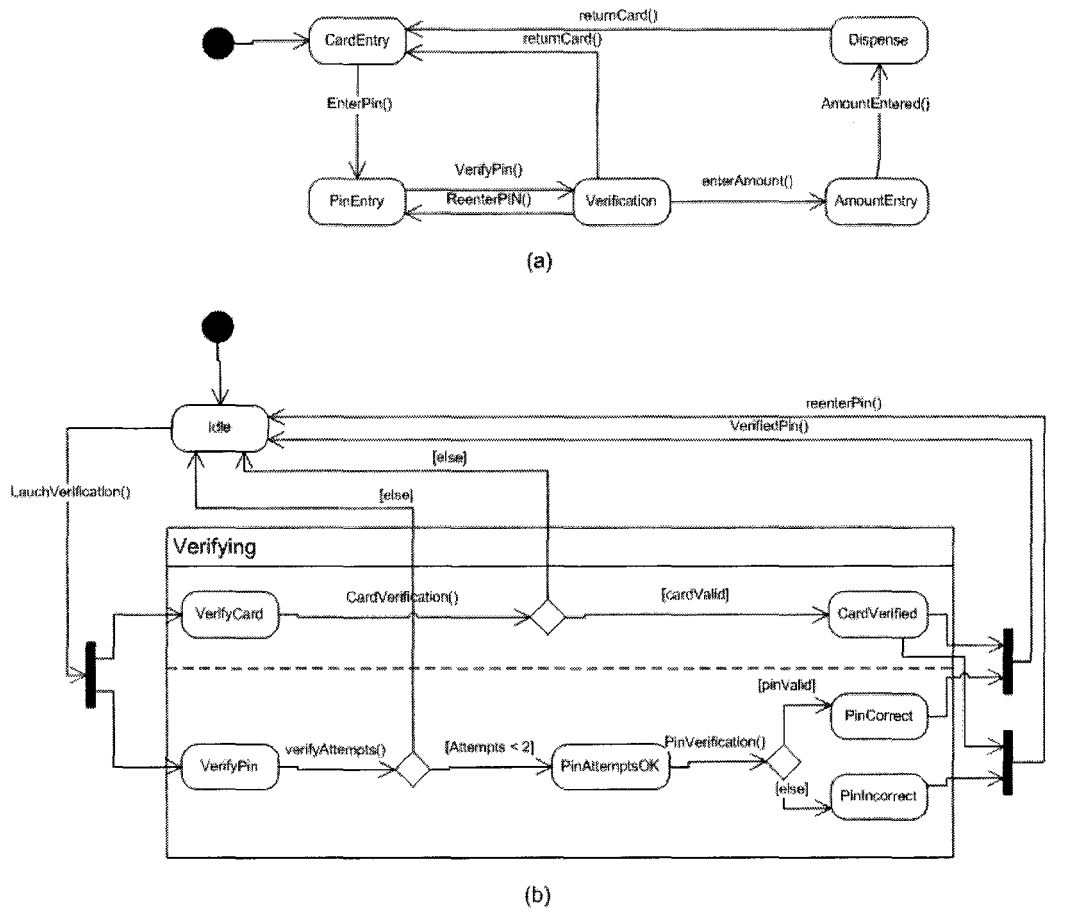


FIG. 5.19 – Diagrammes d'états-transitions du système de guichet automatique, respectivement des classes (a) *ATM* et (b) *Bank*

validité d'une carte bancaire et la validité d'un NIP. Ces deux vérifications peuvent être faites de façon concurrente, tel que le montre le diagramme de communication de la figure 5.20. Dans ce diagramme, les messages *CardVerification* et *verifyAttempts* sont lancés de façon concurrente.

La vérification de la validité d'une carte bancaire est très simple dans le modèle étudié : seulement la valeur de l'attribut *cardValid* est vérifiée. Si l'usager a tenté sans succès d'entrer un NIP valide plus de trois fois, la transaction est automatiquement rejetée. Enfin, la validité du NIP est elle aussi simplifiée dans le modèle étudié : il ne s'agit,

encore une fois, que de vérifier la valeur de l'attribut *pinValid*.

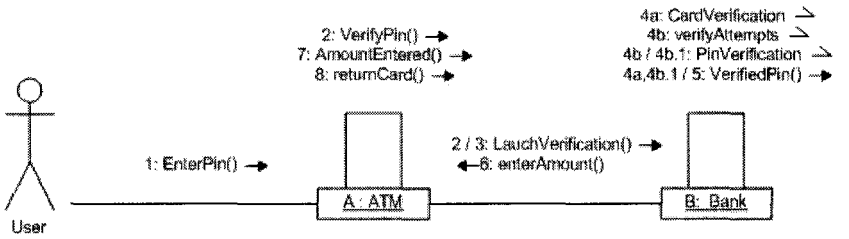


FIG. 5.20 – Diagramme de communication du système de guichet automatique

5.2.2 Translation et Validation

La première partie de cette section applique le processus de translation décrit au chapitre 4, tandis que la seconde montre comment la description formelle ainsi générée peut être vérifiée à l'aide de simulations.

Translation

```
fmod BANK-STATEVALUES is
  sorts BankStateValues BankCardStateValues
        BankPinStateValues .
*** Primary State ************************************
  ops Idle Verifying : -> BankStateValues .
*** Regional Concurrent States ************************
  ops Card-Not-Active VerifyingCard CardValid
        : -> BankCardStateValues .
  ops Pin-Not-Active VerifyingPin PinAttemptsOK
        PinCorrect PinIncorrect : -> BankPinStateValues .
endfm
```

FIG. 5.21 – Le module *BANK-STATEVALUES*

La première étape du processus de translation consiste en la description du nom des états de chacune des classes. Pour ce faire, deux modules fonctionnels sont introduits, *ATM-STATEVALUES* et *BANK-STATEVALUES*. Étant donné la nature concurrentielle de la classe *Bank*, le module *BANK-STATEVALUES* est montré à la figure 5.21.

```
mod BANK is
  protecting METHOD BOOL INT .
  protecting BANK-STATEVALUES .
  *** Class ****************************
  sort Bank .
  subsort Bank < Cid .
  op Bank : -> Bank .
  *** Attributes ***********************
  op StateB :_ : BankStateValues -> Attribute .
  op StateB-Pin :_ : BankPinStateValues -> Attribute .
  op StateB-Card :_ : BankCardStateValues -> Attribute .
  op cardValid :_ : Bool -> Attribute .
  op pinValid :_ : Bool -> Attribute .
  op Attempts :_ : Int -> Attribute .
  *** Methods **************************
  op LaunchVerification :  ParameterList -> Void .
  op VerifyAttempts : ParameterList -> Void .
  op CardVerification : ParameterList -> Void .
  op PinVerification : ParameterList -> Void .
  op VerifiedPin : ParameterList -> Void .
  op reenterPin : ParameterList -> Void .
endm
```

FIG. 5.22 – Le module *BANK*

Le diagramme de classes de la figure 5.18 montre que le système étudié est composé de deux classes : *ATM* et *Bank*. Seul le module *BANK* est donné à la figure 5.22, étant donné que la classe *Bank* montre des aspects concurrentiels. Les deux attributs *StateB-Pin* et *StateB-Card* sont plus particulièrement utilisés pour représenter les deux régions orthogonales du diagramme d'états-transitions de la classe *Bank*.

Le prochain module généré est le module *IDENTIFICATION*. Ce module a pour objectif d'introduire un mécanisme adéquat d'identification des objets, tel qu'il est possible de le voir à la figure 5.23. Deux définitions de types y sont introduits : *Aoid* et *Boid*, qui décrivent respectivement les mécanismes d'identification des objets *A* et *B* des classes *ATM* et *Bank*.

Le module *COMMUNICATION* est le dernier module généré lors de l'application du processus de translation. Par contre, il en est aussi le module principal. Il introduit le comportement individuel et collectif des objets en considérant à la fois les diagrammes d'états-

```
mod IDENTIFICATION is
  including CONFIGURATION .
  protecting STRING .
  sorts Aoid Boid Receiver .
  subsort Aoid Boid < Oid .
  subsort Receiver < Aoid Boid .
  subsort String < Aoid Boid .
endm
```

FIG. 5.23 – Le module *IDENTIFICATION*

transitions associés à chacune des classes présentées à la figure 5.19 et le diagramme de communication représentant l'approche utilisée pour accomplir une tâche donnée. Ce diagramme est présenté à la figure 5.20. La figure 5.24, quant à elle, montre une partie du module *COMMUNICATION*.

La règle de réécriture *[B1]* démarre la vérification en parallèle de la carte bancaire et du NIP. On remarque ceci puisque les attributs d'états régionaux *StateB-Pin* et *StateB-Card* ne sont plus inactifs. La règle *[B2]* montre, quant à elle, comment se passe la vérification de la validité de la carte bancaire. La règle est conditionnelle et elle est gardée par la valeur de l'attribut *cardValid*. Enfin, la règle *[B5]* montre comment le message *VerifiedPin* est envoyé, signifiant que l'usager a été correctement identifié. Les deux messages *IsAccomplished* de la première partie entrent en jeu pour s'assurer que les deux vérifications concurrentes de la carte bancaire et du NIP sont terminées.

La translation du système du guichet automatique vers une notation formelle Maude est maintenant complète. Le module *COMMUNICATION* constitue donc un modèle formel du système dans lequel les aspects statiques et dynamiques ont été pris en considération. La première vérification qui sera tentée pour ce système sera faite à l'aide de simulations.

Validation

Trois vérifications différentes seront effectuées ici : les deux premières sur le comportement de l'objet *A* et de l'objet *B*. La troisième vérification consistera en une simulation du système global.

```
mod COMMUNICATION is
  protecting IDENTIFICATION CORRESPONDENCE .
...
  *** Bank's Behaviour ************************************
  rl [B1]: IsAccomplished(VerifyPin( EmptyParameterList ), A)
    ComingMsg(LaunchVerification( EmptyParameterList ), B)
    < B : Bank | StateB : Idle, StateB-Pin : Pin-Not-Active,
          StateB-Card : Card-Not-Active, cardValid : Card,
          pinValid : Pin, Attempts : pinAtt >
  =>
    < B : Bank | StateB : Verifying, StateB-Pin : VerifyingPin,
          StateB-Card : VerifyingCard, cardValid : Card,
          pinValid : Pin, Attempts : pinAtt > .
  crl [B2]: ComingMsg(CardVerification( EmptyParameterList ), B)
    < B : Bank | StateB : Verifying, StateB-Pin : PinStatus,
          StateB-Card : VerifyingCard, cardValid : Card,
          pinValid : Pin, Attempts : pinAtt >
  =>
    < B : Bank | StateB : Verifying, StateB-Pin : PinStatus,
          StateB-Card : CardValid, cardValid : Card, pinValid :
          Pin, Attempts : pinAtt >
    IsAccomplished(CardVerification( EmptyParameterList ), B)
  if Card == true .
...
  rl [B5]: ComingMsg(VerifiedPin( EmptyParameterList ), B)
    IsAccomplished(CardVerification( EmptyParameterList ), B)
    IsAccomplished(PinVerification (EmptyParameterList ), B)
    < B : Bank | StateB : Verifying, StateB-Pin : PinCorrect,
          StateB-Card : CardValid, cardValid : Card, pinValid :
          Pin, Attempts : pinAtt >
  =>
    < B : Bank | StateB : Idle, StateB-Pin : Pin-Not-Active,
          StateB-Card : Card-Not-Active, cardValid : Card,
          pinValid : Pin, Attempts : pinAtt >
    IsAccomplished(VerifiedPin( EmptyParameterList), B) .
...
endm
```

FIG. 5.24 – Le module *COMMUNICATION*

Vérification du comportement individuel. Les deux premières vérifications effectuées portent respectivement le comportement individuel des objets *A* et *B*. La configuration ini-

tiale de la figure 5.25 est conçue pour vérifier le comportement individuel de l'objet *A*. Elle consiste en l'objet *A* dans son état d'entrée du NIP (*PinEntry*). En recevant le message *VerifyPin*, cet objet devrait alors transiter vers son état *Verification*.

```
ComingMsg(VerifyPin( EmptyParameterList ), A)
< A : Atm | StateA : PinEntry >
```

FIG. 5.25 – Une configuration initiale pour vérifier le comportement de l'objet *A*

Les résultats de cette première simulation sont donnés à la figure 5.26. Ainsi, à la réception du message *ComingMsg(VerifyPin(EmptyParameterList), A)*, l'objet *A* entre dans sa phase de vérification.

```
< A : Atm | StateA : Verification >
IsAccomplished(VerifyPin( EmptyParametersList ), A)
```

FIG. 5.26 – Résultats de la simulation de la figure 5.25

La deuxième vérification concerne tout particulièrement la vérification en parallèle de la carte bancaire et du NIP d'un usager. Pour ce faire, la configuration initiale de la figure 5.27 est introduite. Elle comporte un objet *B* de la classe *Bank* recevant les messages *CardVerification*, *verifyAttempts* et *PinVerification*.

```
ComingMsg(CardVerification( EmptyParameterList ), B)
ComingMsg(VerifyAttempts( EmptyParameterList ), B)
ComingMsg(PinVerification( EmptyParameterList ), B)
< B : Bank | StateB : Verifying, StateB-Pin :
      VerifyingPin, StateB-Card : VerifyingCard, cardValid :
      true, pinValid : true, Attempts : 0 >
```

FIG. 5.27 – Une configuration initiale pour vérifier le comportement de l'objet *B*

Les résultats de cette deuxième simulation sont donnés à la figure 5.28. On y constate alors que les deux vérifications se sont complétées adéquatement. En effet, les attributs d'états régionaux confirment la validité de la carte bancaire et du NIP de l'usager (ils sont dans leurs états *CardValid* et *PinCorrect* respectivement). Les deux messages *IsAccomplished* présents dans ces résultats seront utilisés pour déclencher la règle *[B5]* de la figure 5.24.

```
IsAccomplished(CardVerification(EmptyParameterList), B)
IsAccomplished(PinVerification(EmptyParameterList), B)
< B : Bank | StateB : Verifying,StateB-Pin : PinCorrect,
             StateB-Card : CardValid,cardValid : true,
             pinValid : true,Attempts : 0 >
```

FIG. 5.28 – Résultats de la simulation de la figure 5.27

Vérification du comportement collectif. Les règles de réécritures individuelles ont été vérifiées précédemment. Cependant, est-ce que toutes ces règles, lorsque mises en commun, fonctionnent correctement ? En d'autres termes, est-ce que le système (globalement) fonctionne de façon adéquate ? Afin de vérifier ceci, une simulation du système entier serait pertinente.

```
< A : Atm | StateA : CardEntry >
< B : Bank | StateB : Idle, StateB-Pin :
             Pin-Not-Active, StateB-Card :
             Card-Not-Active, cardValid : true,
             pinValid : true, Attempts : 0 >
ComingMsg(EnterPin( EmptyParameterList ), A)
ComingMsg(VerifyPin( EmptyParameterList ), A)
ComingMsg(returnCard( EmptyParameterList ), A)
ComingMsg(enterAmount( EmptyParameterList ), A)
ComingMsg(AmountEntered( EmptyParameterList ), A)
ComingMsg(LaunchVerification(EmptyParameterList), B)
ComingMsg(CardVerification( EmptyParameterList ), B)
ComingMsg(VerifyAttempts( EmptyParameterList ), B)
ComingMsg(PinVerification( EmptyParameterList ),B)
ComingMsg(VerifiedPin( EmptyParameterList ), B) .
```

FIG. 5.29 – Configuration initiale pour la simulation du système entier

Pour se faire, deux simulations différentes seront faites. Seule la configuration initiale est donnée dans la figure 5.29. Les deux simulations sont toutes les deux démarrées de cette configuration initiale. La raison pour laquelle deux simulations distinctes sont utilisées s'explique par ce qui suit.

– La première réécriture, limitée à cinq pas, servira à visualiser l'état intermédiaire du

système ;

– La seconde réécriture, illimitée, permettra quant à elle de vérifier l'état final du système.

Cette configuration initiale comporte les deux objets *A* et *B*, respectivement des classes *ATM* et *Bank*, dans leurs états initiaux. Il y apparaît également la séquence de messages montrée dans le diagramme de communication de la figure 5.20.

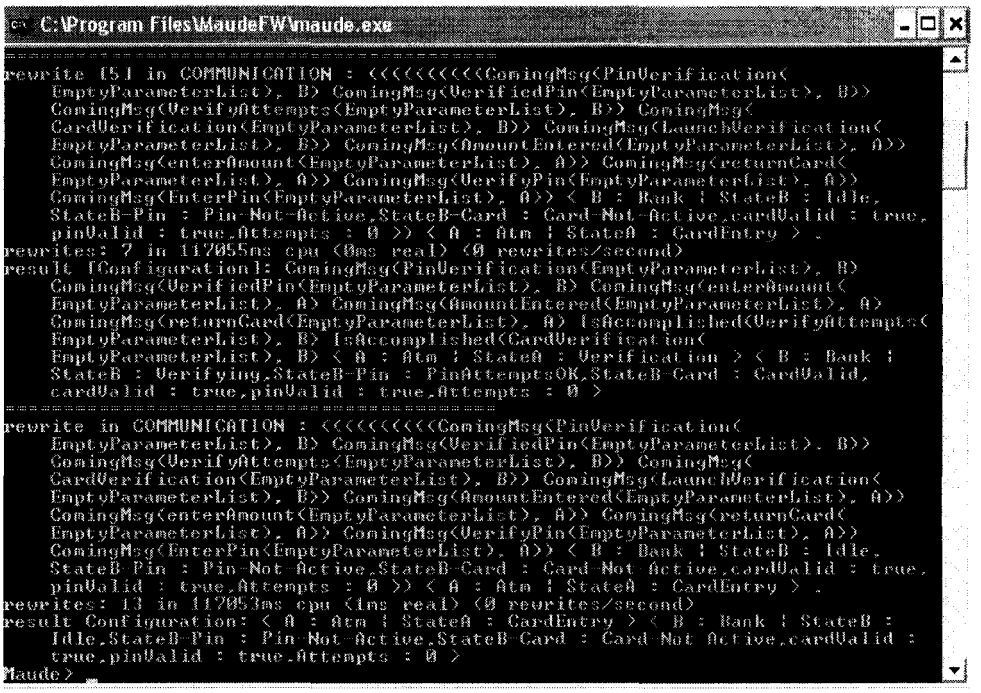


FIG. 5.30 – Résultats de la simulation du système de guichet automatique

Les résultats de ces deux simulations sont donnés à la figure 5.30. La première réécriture permet de vérifier l'état intermédiaire du système. On y remarque alors que l'objet *A* est dans son état *Verification*, alors que l'objet *B*, quant à lui, est dans son état composite *Verifying*. Les régions concurrentes, quant à elles, montrent que la carte bancaire a été vérifiée, alors que le processus de vérification du NIP est encore en cours.

Enfin, la deuxième simulation permet de retrouver le système avec seulement les deux objets *A* et *B* dans leurs états initiaux respectifs. Ceci peut s'interpréter du fait que la transaction de l'usager est terminée et que le guichet automatique est maintenant en attente qu'un autre

usager l'utilise. Cette configuration finale est donc cohérente pour un système de ce type.

5.2.3 Vérification Formelle

Les deux sections qui suivent introduiront respectivement une série de propriétés du comportement individuel et collectif des objets du système. Par la suite, une section décrira plus en détail le processus adopté pour procéder à la vérification, et la dernière partie verra à analyser les résultats obtenus.

Propriétés du comportement individuel

Dans ce qui suit, six propriétés sont proposées pour procéder à la vérification des objets du système. Trois propriétés vérifieront le comportement de l'objet *A*, et quatre autres le comportement de *B*.

Propriété 1 :

- Formule LTL :

```
ATM-In-PinEntry-State("A") ->
    O (ATM-In-Verification-State("A"))
```

- Description : En démarrant la vérification à partir de l'état initial *initial1* (figure 5.32), cette propriété exprime le fait que si le guichet automatique est dans son état *PinEntry*, alors la propriété *ATM-In-Verification-State* exprimant que le guichet automatique se trouve dans son état *Verification*, sera vraie dans son prochain état.

Propriété 2 :

- Formule LTL :

```
ATM-In-CardEntry-State("A") ->
    <> (ATM-In-Verification-State("A"))
```

- Description : En démarrant la vérification à partir de l'état initial *initial1* (figure 5.32), cette propriété exprime le fait que lorsque le guichet automatique est dans son état *CardEntry*, la propriété *ATM-In-Verification-State* exprimant le fait que le guichet automatique soit dans son état *Verification*, sera éventuellement vraie.

Propriété 3 :

- Formule LTL :

```
ATM-In-Verification-State("A")
    |-> ATM-In-Dispense-State("A")
```

- Description : En démarrant la vérification à partir de l'état initial *initial1* (figure 5.32), cette propriété vérifie que lorsque le guichet automatique est dans son état *Verification*, alors la propriété *ATM-In-Dispense-State* exprimant le fait que le guichet automatique est en mode de distribution de billets (son état *Dispense*), sera vraie à un certain moment.

Propriété 4 :

- Formule LTL :

```
Bank-In-Idle-State("B") ->
    O (Bank-In-Verification-VC-VP-State("B"))
```

- Description : En démarrant la vérification à partir de l'état initial *initial1* (figure 5.32), si la banque est dans son état *Idle*, alors la propriété *Bank-In-Verification-VC-VP-State* exprimant que la banque est dans son état *Verifying* (pour lequel les sous-état de ce super état sont dans leurs valeurs initiales respectives, soit *VerifyingCard* et *VerifyingPin*) est vraie dans le prochain état. On vérifie ici que le démarrage des vérifications en parallèle de la carte bancaire et du NIP de l'usager se passe correctement.

Propriété 5 :

- Formule LTL :

```
Bank-In-Verification-CV-PC-State("B") ->
    O (Bank-In-Idle-State("B"))
```

- Description : En démarrant la vérification à partir de l'état initial *initial1* (figure 5.32), cette propriété exprime le fait que si la banque est dans son état *Verification*, alors la propriété *Bank-In-Idle-State* exprimant le fait que la banque est dans son état *Idle* est vraie dans le prochain état. La vérification faite ici consiste en ce que la

transition de la banque vers son état *Idle* après avoir identifié correctement l'usager avec sa carte bancaire et son NIP se fait correctement.

Propriété 6 :

- Formule LTL :

```
Bank-In-Verification-VC-VP-State("B")
    |-> Bank-In-Verification-CV-PC-State("B")
```

- Description : Cette propriété vérifie qu'en démarrant la vérification de l'état initial *initial1* (figure 5.32) et que si la banque est dans son état initial de vérification avec les deux régions orthogonales dans leur état initial respectif (*Verifying – Verifying-Card – VerifyingPin*), alors la propriété *Bank-In-Verification-CV-PC-State* exprimant que la banque est dans son état de vérification où les deux régions orthogonales sont dans leur état final respectif (la carte et le NIP ayant été vérifiés, *CardValid* et *Pin-Valid* respectivement) sera vraie à un certain moment dans le futur.

Propriétés du comportement collectif

Les six prochaines propriétés visent à vérifier le comportement collectif des objets A et B.

Propriété 7 :

- Formule LTL :

```
<> (ATM-In-AmountEntry-State("A") /\
    Bank-In-Verification-X-X-State("B"))
```

- Description : En démarrant la vérification à partir de l'état initial *initial1* (figure 5.32), cette propriété est éventuellement vraie : le guichet automatique est dans son état *AmountEntry* au même moment où la banque est dans son état *Verifying* (pour lequel aucune distinction n'est faite au niveau des régions orthogonales). Noter que cette propriété est non désirable au sein du système étudié.

Propriété 8 :

- Formule LTL :

```
[] (ATM-In-Verification-State("A") ->
```

```
Bank-In-Verification-X-X-State("B"))
```

- Description : En démarrant la vérification à partir de l'état initial *initial1* (figure 5.32), cette propriété est toujours vraie : le guichet automatique est dans son état *Verification* implique que la banque soit dans son état *Verifying*. Cette propriété vient s'assurer de la concordance des états des deux objets lors du processus d'identification d'un usager.

Propriété 9 :

- Formule LTL :

```
[] (ATM-In-PinEntry-State("A")
    -> Bank-In-Idle-State("B"))
```

- Description : En démarrant la vérification à partir de l'état initial *initial1* (figure 5.32), cette propriété est toujours vraie : le guichet automatique est dans son état *PinEntry* implique que la banque soit dans son état *Idle*. Cette propriété vient s'assurer de la concordance des états des deux objets lorsque l'usager débute son utilisation du guichet automatique.

Propriété 10 :

- Formule LTL :

```
[] (ATM-In-AmountEntry-State("A")
    -> Bank-In-Idle-State("B"))
```

- Description : En démarrant la vérification à partir de l'état initial *initial1* (figure 5.32), cette propriété est toujours vraie : le guichet automatique est dans son état *AmountEntry* implique que la banque soit dans son état *Idle*. Cette propriété vient s'assurer de la concordance des états des deux objets lorsque la procédure d'identification de l'usager est complétée et que le guichet automatique est en train de distribuer les billets nécessaires.

Propriété 11 :

- Formule LTL :

```
[] (Coherent-System-State("A", "B"))
```

TAB. 5.3 – Les états cohérents du système de guichet automatique

Classe *ATM*	**Classe *Bank***		
	Principal	Régions Orthogonales	
		Carte	NIP
CardEntry	Idle	Card-Not-Active	Pin-Not-Active
PinEntry	Idle	Card-Not-Active	Pin-Not-Active
AmountEntry	Idle	Card-Not-Active	Pin-Not-Active
Dispense	Idle	Card-Not-Active	Pin-Not-Active
Verification	Idle	Card-Not-Active	Pin-Not-Active
Verification	Verifying	VerifyingCard	VerifyingPin
Verification	Verifying	VerifyingCard	PinAttemptsOK
Verification	Verifying	VerifyingCard	PinCorrect
Verification	Verifying	VerifyingCard	PinIncorrect
Verification	Verifying	CardValid	VerifyingPin
Verification	Verifying	CardValid	PinAttemptsOK
Verification	Verifying	CardValid	PinCorrect
Verification	Verifying	CardValid	PinIncorrect

- Description : En démarrant la vérification à partir de l'état initial *initial2* (figure 5.32), cette propriété exprime le fait que le système est toujours dans un état dit cohérent. Le tableau 5.3 montre les états correspondants formant des états globaux du système qui sont cohérents.

Propriété 12 :

- Formule LTL :

```
[] (Coherent-System-State("A", "B"))
```

- Description : Cette propriété exprime le fait qu'en démarrant la vérification à partir d'un état initial *initial3* (figure 5.32), le système sera toujours dans un état dit cohérent. L'état initial *initial3* ne figure cependant pas dans le tableau 5.3 des états globaux cohérents.

La figure 5.31 présente une partie du module *COMMUNICATION-PREDICATES*. Les prédicats montrés dans la figure sont limités aux trois prédicats relatifs à l'objet *A* de la classe *ATM*. Les lignes 1, 2 et 3 sont les prédicats définis pour les états *CardEntry*, *PinEntry* et *Verification* respectivement. Le prédicat *Coherent-System-State("A", "B")* (ligne 4) est

```
mod COMMUNICATION-PREDICATES is
  protecting COMMUNICATION .
  including SATISFACTION .
  subsort Configuration < State .
  ...
  *** Operators ****************************************
  ops ATM-In-CardEntry-State ATM-In-PinEntry-State
      ATM-In-Verification-State ATM-In-AmountEntry-State
      ATM-In-Dispense-State : Aoid -> Prop .
  ops Bank-In-Idle-State Bank-In-Verification-VC-VP-State
      Bank-In-Verification-VC-PAO-State
      Bank-In-Verification-VC-PC-State
      ...
      Bank-In-Verification-X-X-State : Boid -> Prop .
  *** ATM's Behavior ***********************************
  eq < A : Atm | StateA : CardEntry > conf
     |= ATM-In-CardEntry-State( "A" ) = true .      ***[1]
  eq < A : Atm | StateA : PinEntry > conf
     |= ATM-In-PinEntry-State( "A" ) = true .       ***[2]
  eq < A : Atm | StateA : Verification > conf
     |= ATM-In-Verification-State( "A" ) = true .   ***[3]
  ...
  *** System in a Coherent State ************************
  ceq < A : Atm | StateA : AS > < B : Bank | StateB : BS,
        StateB-Pin : BSP, StateB-Card : BSC, cardValid :
        card, pinValid : pin, Attempts : att > conf
    |= Coherent-System-State( "A", "B" ) = true     ***[4]
    if CorrespondingState(AS, (BS, (BSC, BSP))) == true .
  ***************************************************
  ...
endm
```

FIG. 5.31 – Une partie du module *COMMUNICATION-PREDICATES*

également présenté étant donné sa nature différente des autres. Ce prédicat est conditionnel au résultat obtenu à l'évaluation de la fonction *CorrespondingStates*. Cette fonction retourne un résultat positif (*True*) dans les cas énumérés dans le tableau 5.3.

```
mod COMMUNICATION-CHECK is
  including COMMUNICATION-PREDICATES .
  including MODEL-CHECKER LTL-SIMPLIFIER .
  *** Operators ****************************************
  ops initial1 initial2 initial3 : -> Configuration .
  *** Initial Configurations ***************************
  eq initial1 = < "A" : Atm | StateA : CardEntry >
     < "B" : Bank | StateB : Idle, StateB-Pin :
       Pin-Not-Active, StateB-Card : Card-Not-Active,
       cardValid : true, pinValid : true, Attempts : 0 >
    ComingMsg(EnterPin( EmptyParameterList ), "A")
    ComingMsg(VerifyPin( EmptyParameterList ), "A")
    ComingMsg(returnCard( EmptyParameterList ), "A")
    ComingMsg(enterAmount( EmptyParameterList ), "A")
    ComingMsg( AmountEntered( EmptyParameterList ), "A")
    ComingMsg(LaunchVerification( EmptyParameterList ),
       "B")                                          ***[1]
    ComingMsg(CardVerification(EmptyParameterList ), "B")
    ComingMsg(VerifyAttempts( EmptyParameterList ), "B")
    ComingMsg(PinVerification( EmptyParameterList ), "B")
    ComingMsg( VerifiedPin( EmptyParameterList ), "B") .
  eq initial2 = < "A" : Atm | StateA : AmountEntry >
     < "B" : Bank | StateB : Idle, StateB-Pin :     ***[2]
     Pin-Not-Active, StateB-Card : Card-Not-Active,
     cardValid : true, pinValid : true, Attempts : 0 > .
  eq initial3 = < "A" : Atm | StateA : AmountEntry >
     < "B" : Bank | StateB : Verifying, StateB-Pin :
     VerifyingPin, StateB-Card : VerifyingCard, cardValid :
     true, pinValid : true, Attempts : 0 > .       ***[3]
endm
```

FIG. 5.32 – Le module *COMMUNICATION-CHECK*

Vérification des Propriétés

Lorsque les 12 propriétés ont été énoncées, il a été question de plusieurs états initiaux par lesquels doit débuter la vérification de chacune d'entre elles. En plus d'introduire ces configurations initiales, le module *COMMUNICATION-CHECK*, montré à la figure 5.32, établit le lien final entre le système étudié et le vérificateur de modèles de Maude. Seulement trois configurations initiales sont utilisées dans la vérification des 12. La ligne 1 montre l'état

initial utilisé pour la vérification des dix premières propriétés, tandis que les lignes 2 et 3 font expressément référence aux Propriétés 11 et 12 respectivement, concernant la cohérence de l'état global du système.

```
red modelCheck(initial1, ATM-In-PinEntry-State("A")
       -> O (ATM-In-Verification-State("A"))) .
red modelCheck(initial1, ATM-In-CardEntry-State("A")
      -> <> (ATM-In-Verification-State("A"))) .
... red modelCheck(initial1, [] (ATM-In-AmountEntry-State("A")
                -> Bank-In-Idle-State("B"))) .
red modelCheck(initial2,
        [] (Coherent-System-State("A", "B"))) .
...
```

FIG. 5.33 – Quelques appels à la fonction *modelCheck*

La figure 5.33 montre quelques-uns des appels à la fonction *modelCheck* qui lance la vérification de modèles sur le système. Une partie des résultats de l'évaluation de ces 12 propriétés est montrée dans la capture d'écran de la figure 5.34. De plus, un résumé des résultats est donné par le tableau 5.4, où l'on retrouve, pour chacune des propriétés, un résultat positif ou un résultat de contre-exemple.

Analyse des résultats et Conclusion

Les deux sections suivantes analysent respectivement les résultats obtenus pour la vérification du comportement individuel et le comportement collectif des objets du système de guichet automatique.

Résultats du comportement individuel. Les Propriétés 1, 2 et 3 tentent de vérifier le comportement interne de la classe *ATM* tandis que les Propriétés 4, 5 et 6 tentent la même chose pour la classe *Bank*. Ces six propriétés ont toutes donné un résultat positif (*True*) lors de leur vérification.

Il semblerait que le comportement de la classe *ATM* soit défini adéquatement. En effet, la Propriété 1 exprime le fait que lorsque le guichet automatique se trouve dans son état

TAB. 5.4 – Résultats de l'évaluation des 12 propriétés du système de guichet automatique

Propriété	**Résultat**	**Propriété**	**Résultat**
1	Vrai	7	Contre-exemple
2	Vrai	8	Vrai
3	Vrai	9	Vrai
4	Vrai	10	Vrai
5	Vrai	11	Vrai
6	Vrai	12	Contre-exemple

PinEntry, l'état suivant est son état *Verification*. La Propriété 2, quant à elle, vérifie que lorsque l'objet *A* est dans son état initial *CardEntry*, ce même objet est éventuellement dans son état *Verification*. Enfin, la Propriété 3, vérifie que lorsque l'objet *A* est dans son état *Verification*, il est, à un certain moment dans le futur, dans son état *Dispense*. Il faut noter que ce n'est pas toujours le cas en réalité (par exemple, lorsque l'usager annule son opération). Selon le cas étudié ici (voir le diagramme de communication de la figure 5.20), c'est toujours le cas.

D'autre part, le comportement de la classe *Bank* semble lui aussi défini correctement. Les trois propriétés définies visaient à vérifier la séquence des transitions concernant le comportement interne de la classe *Bank* (voir figure 5.19.(b)). La Propriété 4 vérifie que l'objet *B* passe son état *Idle* à son état *Verifying* correctement, tout en s'assurant de la validité des régions orthogonales concurrentes de ce dernier état, sachant qu'à l'activation de l'état composite *Verifying*, les deux régions orthogonales s'activent et prennent leurs valeurs initiales respectives, soit *VerifyingCard* et *VerifyingPin*. En réalité, cette propriété vient s'assurer que la structure "*Fork*" du diagramme d'états-transitions de la figure 5.19.(b) s'active correctement. La Propriété 5 vérifie l'inverse : que les conditions de retour à l'état *Idle* étaient respectées (conditions de la structure "*Join*" des deux régions orthogonales de l'état composite *Verifying*). Ces conditions sont simples : la carte bancaire et le NIP de l'usager doivent tous

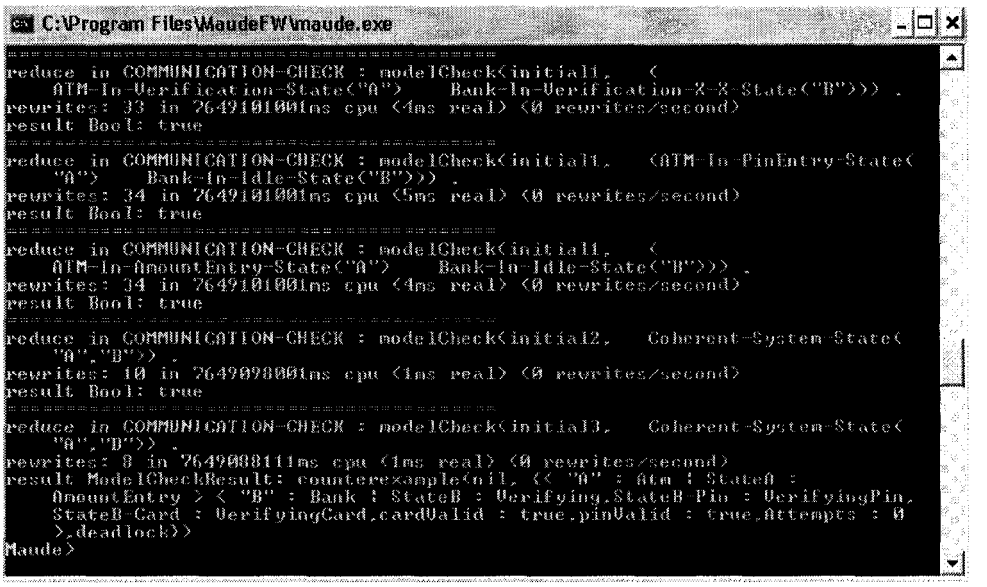


FIG. 5.34 – Partie des résultats des appels à la fonction *modelCheck*

les deux avoir été validés. Enfin, la Propriété 6 s'assure que les deux vérifications, soit la validation de la carte bancaire et du NIP de l'usager, s'effectuent bel et bien en parallèle et que ces derniers sont vérifiés à un certain moment dans le futur.

Résultats du comportement collectif. Les Propriétés 7 à 12 ont pour objectif de valider le comportement collaboratif des objets *A* et *B*. De ces six propriétés, les Propriétés 7 et 12 ont obtenues un résultat négatif alors que toutes les autres ont obtenues un résultat positif. Les résultats peuvent s'interpréter comme suit.

Pour la Propriété 7, pour laquelle l'objectif est de vérifier s'il est possible pour le système de permettre qu'un objet de la classe *ATM* puisse être en mode de distribution d'argent (son état *AmountEntry* par laquelle débute la distribution d'argent), alors que la vérification de la carte bancaire et du NIP de l'usager n'est pas encore complétée (l'objet *B* est encore dans son état *Verifying*), le système retourne un contre-exemple. En effet, le système arrive au bout des réécritures qu'il peut faire sans toutefois satisfaire la propriété étudiée. Donc, cette propriété n'est pas possible au sein du système, ce qui est correct puisqu'elle était en soi non désirable.

Les Propriétés 8 à 10 ont quant à elles été vérifiées. Ces trois propriétés ont été dévelop-

pées en vue de prouver la concordance des états entre les deux objets lors des diverses phases du processus de retrait par un usager. La Propriété 8 vérifie plus particulièrement qu'il est toujours vrai que lorsque le guichet est en phase de vérification, cela implique que la banque l'est également. La Propriété 9, quant à elle, prouve que si le guichet est à l'étape de demander à l'usager son NIP, la banque, elle, n'a pas débuté la vérification (elle est encore dans son état *Idle*). Enfin, la Propriété 9 vérifie pour sa part que lorsque le guichet entre dans sa phase de distribution de billets, la banque a terminé la phase de vérification (elle est revenue dans son état *Idle*).

Les deux dernières propriétés, les Propriétés 11 et 12, ont pour objectif de vérifier l'état global du système. Pour ce faire, les états correspondants formant des états globaux cohérents du système ont été donnés dans le tableau 5.3. La Propriété 11 a été vérifiée à partir d'un état cohérent du système et a obtenu un résultat positif. Quant à elle, la Propriété 12 a été vérifiée à partir d'un état incohérent du système. Elle a obtenu un résultat de contre-exemple, comme le montre la figure 5.34. Il est donc possible de conclure que les états globaux du système fonctionnent.

5.3 Le Système du Producteur – Consommateur

Le troisième exemple étudié est un problème classique du monde de l'informatique : le problème du Producteur et du Consommateur. Deux objets concurrents, un Producteur produisant des éléments d'information à transmettre, et un Consommateur tentant d'accéder à ces informations, veulent tous les deux accéder à une ressource partagée en même temps. Ce problème présente des caractéristiques de concurrence externe. Il est étudié ici pour démontrer que l'approche proposée s'applique sur un problème de concurrence externe.

5.3.1 Présentation

Le cas présenté ici est un simple programme montrant des caractéristiques d'un système orienté-objet concurrentiel (SOOC). Ce petit système est basé sur le problème classique du Producteur – Consommateur. Le système présenté ici est une adaptation des problèmes similaires présentés par Gomaa et Meseguer dans leurs publications respectives [35, 64]. La figure 5.35 présente le diagramme de classes de ce système.

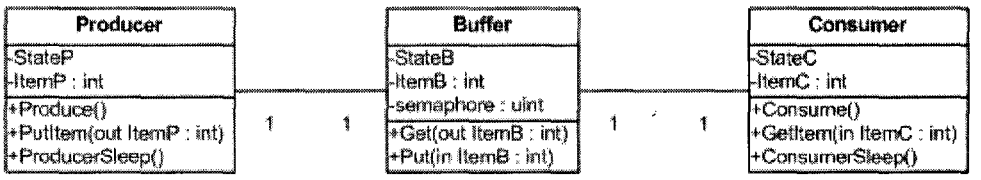


FIG. 5.35 – Diagramme de classes du système du Producteur – Consommateur

La classe *Producer* : Cette classe présente des objets actifs ayant pour objectif de générer des éléments d'information sous la forme de nombres entiers. Cette production se fera au travers de la fonction *Produce*. Les objets de classe *Producer* auront également à transmettre leur information à la ressource partagée via la fonction *PutItem*. Enfin, cette classe a aussi une fonction *ProducerSleep* qui mettra en attente les objets de cette classe lorsque la ressource partagée ne sera pas disponible pour recevoir de nouvelles informations. Cette classe a deux attributs. Le premier, *StateP*, est un attribut qui repré-

sentera l'état actuel d'un objet de type *Producer*, tandis que l'attribut *ItemP* sera utilisé pour stocker la valeur d'une information de type entière à transmettre ;

La classe *Consumer* : Cette classe présente des objets actifs ayant pour objectif d'aller chercher de l'information sous forme de nombres entiers au sein d'une ressource partagée. Cette consommation d'information se fera via la fonction *GetItem*. Les objets de cette classe accomplissent par la suite une tache quelconque avec les informations obtenues via la fonction *Consume*. Enfin, cette classe dispose également d'une fonction *ConsumerSleep* mettant en attente le processus des objets de cette classe lorsque la ressource partagée n'est pas disponible pour la transmission de nouvelles informations. Cette classe dispose de deux attributs. *StateC*, sera utilisé pour représenter l'état actuel d'un objet de classe *Consumer*, tandis que *ItemC* sera utilisé pour stocker l'information qu'un objet de type *Consumer* aura été chercher au sein de la ressource partagée ;

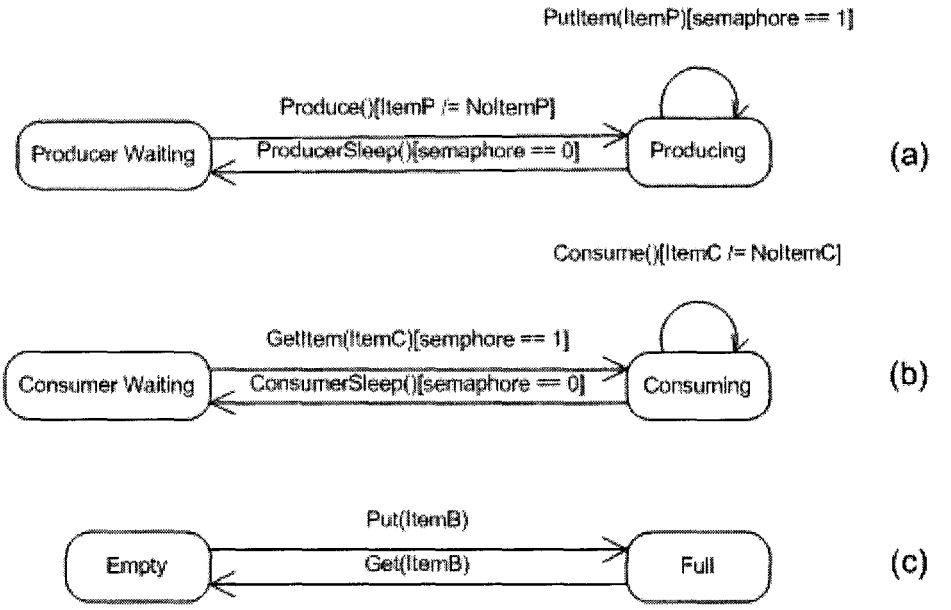


FIG. 5.36 – Diagrammes d'états-transition du système du Producteur – Consommateur, respectivement des classes (a) *Producer*, (b) *Consumer* et (c) *Buffer*

La classe *Buffer* : Cette classe est en réalité la ressource partagée du système. Elle peut contenir une seule information de type entière à la fois. Les fonctions *Put* et *Get* auront

pour effet de placer une information dans la mémoire partagée et de transmettre cette information respectivement. Cette classe dispose de trois attributs. Le premier, *StateB*, représentera l'état actuel d'un objet de classe *Buffer* (voir la figure 5.36.(c) pour les détails des diverses valeurs d'état possibles). Le second attribut est nommé *ItemB* et représente en réalité la mémoire partagée du système. Cette mémoire partagée est de taille 1 et contient des informations de type entier.

Le troisième attribut de cette classe est nommé *semaphore*. Au niveau des systèmes concurrentiels, un sémaphore est utilisé pour coordonner les accès à une ressource commune à deux ou plusieurs processus concurrents. Les valeurs possibles pour *semaphore* sont 0 et 1 et s'interprète comme suit :

- Lorsque *semaphore* == *1*, la mémoire partagée est libre (aucun autre processus n'y accède à ce moment précis) et peut être utilisée soit par un processus de type Producteur ou de type Consommateur ;
- Lorsque *semaphore* == *0*, la mémoire partagée est actuellement utilisée par un autre objet, et son accès est alors verrouillé, prévenant ainsi tout autre objet d'y accéder alors qu'un processus est actuellement en train de procéder à sa mise à jour.

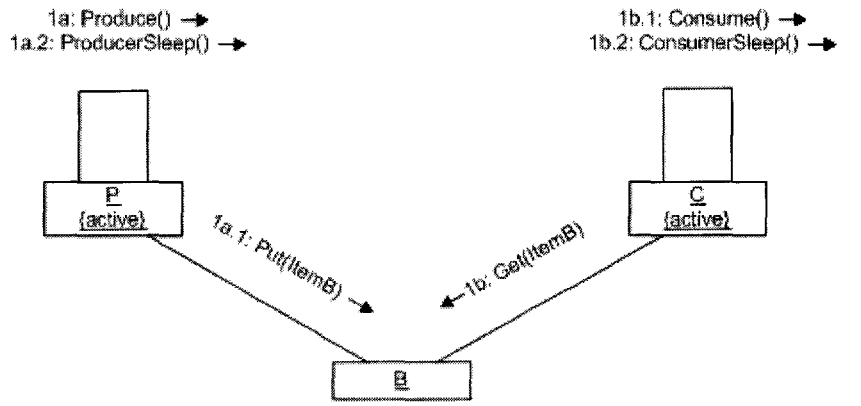


FIG. 5.37 – Diagrammes de communication du système du Producteur – Consommateur

Le diagramme de communication de la figure 5.37 présente un cas typique d'utilisation du problème du Producteur – Consommateur. On y distingue deux objets, *P* et *C* (de classe

Producer et *Consumer* respectivement), tentant d'accéder en même temps à la ressource partagée, un objet *B* de classe *Buffer*. Les fonctions *PutItem* et *GetItem* des objets *P* et *C* étant gardées à l'aide du sémaphore de *B*, l'accès simultané à la ressource commune n'est pas permis. Enfin, comme *P* et *C* sont deux objets indépendants disposant de leur propre processus d'exécution, ils porteront l'appellation d'objet actif (*Active Objet*).

5.3.2 Translation et Validation

La première partie de cette section appliquera le processus de translation décrit au chapitre 4, tandis que la seconde verra comment la description formelle ainsi générée peut être vérifiée à l'aide de simulations.

Translation

Dans cette section, le problème classique du Producteur – Consommateur présenté à la section précédente sera transformé en une notation formelle Maude à l'aide du processus de translation développé.

Comme le système étudié comporte trois classes distinctes, trois modules fonctionnels Maude sont créés, un par classe, pour décrire les états de chacun. Ces trois modules sont nommés *PRODUCER-STATEVALUES*, *CONSUMER-STATEVALUES* et *BUFFER- STATEVALUES*. Ces modules sont créés en rapport avec les classes *Producer*, *Consumer* et *Bufferer* respectivement. La figure 5.38 présente le module *PRODUCER-STATEVALUES*, tandis que les 2 autres modules sont omis.

```
fmod PRODUCER-STATEVALUES is
  sort ProducerStateValues .
  ops Producing ProducerWaiting : -> ProducerStateValues .
endfm
```

FIG. 5.38 – Le module *PRODUCER-STATEVALUES*

Pour chacune des classes, il faudra créer un module portant le nom de cette classe, et dans lequel seront décrites ces dernières, leurs attributs et leurs méthodes. Seulement un de

ces modules, *PRODUCER*, est décrit à la figure 5.39.

```
mod PRODUCER is
  protecting METHOD PRODUCER-STATEVALUES INT .
  sort Producer .
  subsort Producer < Cid .
  *** Class and Attributes
  op Producer : -> Producer .
  op StateP :_ : ProducerStateValues -> Attribute .
  op ItemP :_ : Int -> Attribute .
  *** Methods
  op Produce : ParameterList -> Void .
  op PutItem : ParameterList -> Void .
  op ProducerSleep : ParameterList -> Void .
  op NoItemP : -> Int .
endm
```

FIG. 5.39 – Le module *PRODUCER*

L'élément *NoItemP* est un peu particulier. Il faut le considérer comme un nouvel élément à l'ensemble des nombres entiers. Il sera utilisé en vue de signifier que l'attribut *ItemP* ne contient aucun élément à un certain moment.

Le prochain module généré est le module *IDENTIFICATION*. Ce module a pour objectif d'introduire un mécanisme d'identification des objets adéquat, tel qu'il est possible de le voir à la figure 5.40. Trois définitions de types y sont introduits : *Poid*, *Coid* et *Boid*, qui décrivent respectivement les mécanismes d'identification des objets *P*, *C* et *B* des classes *Producer*, *Consumer* et *Buffer* respectivement.

```
mod IDENTIFICATION is
  including CONFIGURATION .
  protecting STRING .
  sorts Poid Coid Boid Receiver .
  subsort Poid Coid Boid < Oid .
  subsort Receiver < Poid Coid Boid .
  subsort String < Poid Coid Boid .
endm
```

FIG. 5.40 – Le module *IDENTIFICATION*

Le dernier module généré lors du processus de translation de l'exemple du Producteur – Consommateur est le module *COMMUNICATION* montré à la figure 5.41.

```
mod COMMUNICATION is
  protecting IDENTIFICATION PRODUCER CONSUMER BUFFER .
  subsort Int < Parameter .
  *** Utility Messages ********************************
  op ComingMsg : Event Receiver -> Msg .
  op IsAccomplished : Event Receiver -> Msg .
  *** Variables ***************************************
  var P : Poid .   var sema : Nat .
  var C : Coid .   var item : Int .     var B : Boid .
  *** Producer's behavior *****************************
  crl [ProduceItem]:
    ComingMsg(Produce( EmptyParameterList), P)
    < P : Producer | StateP : ProducerWaiting, ItemP : item >
    < B : Buffer | StateB : Empty, ItemB : NoItemB,
                   Semaphore : sema >
   =>
    < P : Producer | StateP : Producing, ItemP : 5 >
    < B : Buffer | StateB : Empty, ItemB : NoItemB,
                   Semaphore : sema >
    IsAccomplished( Produce( EmptyParameterList), P)
    ComingMsg( Put( 5 ), B)
   if item == NoItemP .
  crl [PutItem]:
    ComingMsg( Put( 5 ), B)
    IsAccomplished( Produce( EmptyParameterList), P)
    < P : Producer | StateP : Producing, ItemP : 5 >
    < B : Buffer | StateB : Empty, ItemB : NoItemB,
                           Semaphore : sema >
   =>
    < P : Producer | StateP : Producing, ItemP : NoItemP >
    < B : Buffer | StateB : Full, ItemB : 5, Semaphore : 0 >
    ComingMsg( ProducerSleep( EmptyParameterList ), P)
    IsAccomplished( Put( 5 ), B)
   if sema == 1 .
  ...
endm
```

FIG. 5.41 – Le module *COMMUNICATION*

La partie montrée à la figure 5.41 du module *COMMUNICATION* concerne de façon plus particulière le comportement d'un objet de la classe *Producer*. Deux règles de réécriture sont montrées dans la figure. La règle '*ProduceItem*' montre le comportement de l'objet *P* de la classe *Producer* lorsqu'il reçoit un message '*Produce*' lui demandant de débuter la production d'un élément entier. L'exécution de cette méthode produit un élément entier et le place dans l'attribut *ItemP*. À la fin de l'exécution de cette méthode, un message de synchronisation *IsAccomplished* est également produit afin de permettre l'exécution du second message, '*PutItem*'. Il est également intéressant de noter que cette règle est conditionnelle à ce que l'objet *P* ne contienne pas déjà un élément pour qu'elle puisse être exécutée.

Ce second message fait l'objet de la seconde règle : '*PutItem*'. L'exécution de la méthode *PutItem* transmet l'information entière à l'objet *B* de classe *Buffer* via l'utilisation de sa méthode *Put*. L'exécution de la règle est conditionnelle à ce que le sémaphore de l'objet *B* de classe *Buffer* soit à la valeur 1, signifiant que la ressource commune est disponible. Ici également un message de synchronisation *IsAccomplised* est produit pour permettre l'exécution d'une troisième règle de réécriture non montrée, '*ProducerSleep*' qui aura pour effet de mettre l'objet *P* en veille. À partir de ce moment, l'objet *P* attend que la ressource commune, l'objet *B*, soit de nouveau disponible avant de reprendre ses activités.

Validation de la Description Formelle

Trois vérifications par simulations différentes seront effectuées ici : une première sur le comportement de l'objet *P*, l'autre sur le comportement de l'objet *C* et la dernière sur le système en entier.

Vérification du comportement de *P*. La première vérification qui sera tentée ici est décrite à l'aide de la configuration initiale donnée à la figure 5.42. On tente de vérifier que l'objet *P*, de classe *Producer*, se comporte correctement : il doit produire un élément d'information entière et le transmettre à l'objet *B*, pour ensuite se mettre en veille car la mémoire de la ressource commune est pleine.

Les résultats de cette première simulation sont donnés à la figure 5.43. Suite à l'exécu-

```
< P : Producer | StateP : ProducerWaiting, ItemP : NoItemP >
< B : Buffer | StateB : Empty, ItemB : NoItemB, Semaphore : 1 >
ComingMsg( Produce( EmptyParameterList ), P )
```

FIG. 5.42 – Une configuration initiale pour vérifier le comportement de l'objet *P*

tion de la méthode *Produce*, l'objet *P* a produit un élément entier, ici 5, et l'a transmit à la ressource commune *B*. *P*, suite à l'exécution de sa méthode *ProducerSleep*, est dans son état *ProducerWaiting* et attend la prochaine occasion où *B* sera disponible et vide (*Empty*). Le fait de retrouver un nouveau message entrant (*ComingMsg*) démarrant à nouveau la production d'un élément entier est correct, puisque le problème classique du Producteur – Consommateur est un processus perpétuel.

```
ComingMsg( Produce( EmptyParameterList ), P )
< P : Producer | StateP : ProducerWaiting, ItemP : NoItemP >
< B : Buffer | StateB : Full, ItemB : 5, Semaphore : 1 >
```

FIG. 5.43 – Résultats de la simulation de la figure 5.42

Vérification du comportement de *C*. Au même égard que *P*, *C* a son propre comportement et doit être vérifié. Pour ce faire, la configuration initiale de la figure 5.44 est utilisée. Elle est constituée d'abord d'un objet *B* de classe *Buffer* qui contient déjà un élément d'information (son état est à *Full*). Un objet *C* de classe *Consumer* reçoit alors un message de tenter d'aller chercher l'information de *B*.

```
ComingMsg( Get( 5 ), B )
< C : Consumer | StateC : ConsumerWaiting, ItemC : NoItemC >
< B : Buffer | StateB : Full, ItemB : 5, Semaphore : 1 > .
```

FIG. 5.44 – Une configuration initiale pour vérifier le comportement de l'objet *C*

Les résultats de cette nouvelle simulation sont donnés à la figure 5.45. L'objet *C*, de nouveau en veille, attend le prochain moment où l'objet *B* sera disponible et qu'il contiendra de nouvelles informations (son état sera à *Full*). Quant à lui, l'objet *B* est de nouveau disponible

et sa mémoire est vide (*Empty*). Encore une fois, le fait de retrouver un message entrant (*ComingMsg*) destiné à l'objet *C* lui demandant d'aller chercher l'information contenue dans *B* est normal puisque le problème du Producteur – Consommateur est un processus perpétuel.

```
ComingMsg( Get( 5 ), B )
< B : Buffer | StateB : Empty, ItemB : NoItemB, Semaphore : 1 >
< C : Consumer | StateC : ConsumerWaiting, ItemC : NoItemC >
```

FIG. 5.45 – Résultats de la simulation de la figure 5.44

Vérification du système entier. Maintenant que les deux parties principales du système du Producteur – Consommateur (le Producteur et le Consommateur) ont été vérifiées individuellement, est-ce que les deux ensembles accompliront la tâche désirée ? C'est ce que cette section tente de vérifier.

La figure 5.46 montre la configuration initiale utilisée pour vérifier le système en entier. Afin de tester le comportement du système, deux commandes de réécriture Maude sont données :

- La première réécriture, limitée à quatre pas, servira à visualiser les états intermédiaires des objets du système ;
- La seconde réécriture, limitée à six pas, permettra de visualiser si le système revient à son état original. Rappelons que le problème du Producteur – Consommateur est un cycle qui se répète à l'infini. Il est donc normal de retrouver la configuration initiale à un certain moment.

```
ComingMsg(Produce( EmptyParameterList), P)
ComingMsg( Get(5), C)
< P : Producer | StateP : ProducerWaiting, ItemP : NoItemP >
< C : Consumer | StateC : ConsumerWaiting, ItemC : NoItemC >
< B : Buffer | StateB : Empty, ItemB : NoItemB, Semaphore :
  1 > .
```

FIG. 5.46 – Commandes de réécriture pour vérifier le système entier

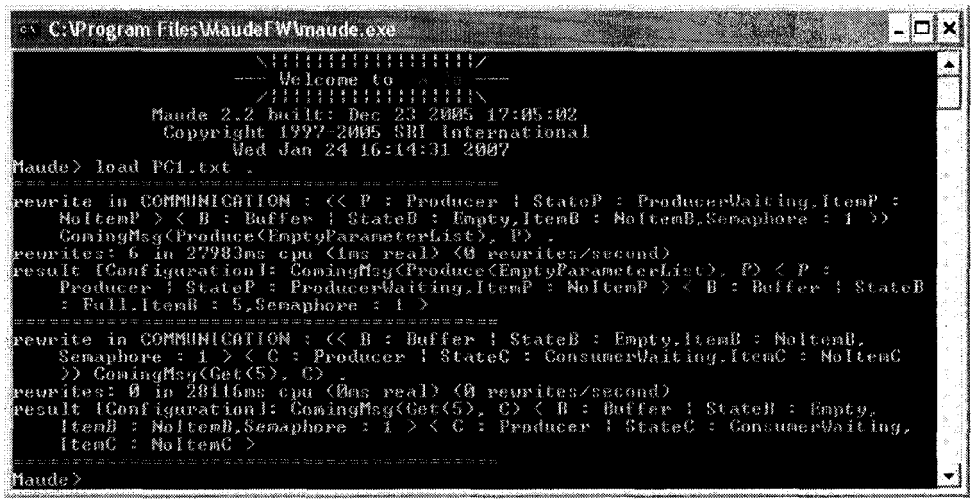


FIG. 5.47 – Résultats de la simulation du système du Producteur – Consommateur

La figure 5.47 montre une capture d'écran de MaudeFW. Les résultats des deux réécritures décrites précédemment y sont montrés. On y remarque que le système revient éventuellement à sa configuration initiale (tel que tentant de vérifier la réécriture limitée à 6 pas). Quant à elle, la première réécriture permet de s'assurer que le système se comporte adéquatement et en concordance avec ce qui avait été déterminé lors des vérifications précédentes.

5.3.3 Vérification Formelle

Les deux sections suivantes introduiront les propriétés qui seront vérifiées au sein du système. La première concernera la vérification du comportement individuel, tandis que la seconde s'attardera au comportement collectif des objets.

Propriétés du comportement individuel

Dans ce qui suit, quatre propriétés sont proposées pour vérifier le comportement individuel des objets P et C. Comme B est un objet outil, son comportement individuel n'est pas vérifié explicitement. Quatre propriétés sont énoncées, deux pour le comportement de P et deux autres pour le comportement de C.

Propriété 1 :

- Formule LTL :

  ```
  [] PutingItem("P")
  ```

- Description :

  En démarrant la vérification à partir de l'état initial *initial1* (voir la figure 5.49), cette propriété exprime le fait que le Producteur est toujours dans sa section critique. Si *P* est toujours dans sa section critique et ne la quitte jamais, le système souffre alors d'un blocage. Cette situation est donc non désirable.

Propriété 2 :

- Formule LTL :

  ```
  ~ [] ProducerSleeping("P")
  ```

- Description :

  En démarrant la vérification à partir de l'état initial *initial1* (voir la figure 5.49), cette propriété vérifie que l'objet Producteur n'est pas toujours dans son état *ProducerWaiting*. Le Producteur n'étant pas toujours dans son état de veille signifie qu'éventuellement il accomplira sa tâche de production d'information. Ceci représente donc une caractéristique désirable du système étudié.

Propriété 3 :

- Formule LTL :

  ```
  [] GetingItem("C")
  ```

- Description :

  En démarrant la vérification à partir de l'état initial *initial1* (voir la figure 5.49), cette propriété vérifie si l'objet Consommateur est toujours dans sa section critique. Le Consommateur toujours dans sa section critique et ne la quittant pas, représente une situation non désirable au sein du système étudié.

Propriété 4 :

- Formule LTL :

  ```
  ~ [] ConsumerSleeping("C")
  ```

- Description :

  En démarrant la vérification à partir de l'état initial *initial1* (voir la figure 5.49), cette propriété exprime le fait que l'objet Consommateur n'est pas toujours en veille (dans son état *ConsumerWaiting*). L'objet *C* quittant son état d'attente signifie qu'éventuellement il accomplira la tâche qui lui est dévolue. Ceci est une caractéristique désirable du système étudié.

Propriétés du comportement collectif

Dans ce qui suit, trois propriétés du comportement collectif des objets *P*, *C* et *B* sont énoncées.

Propriété 5 :

- Formule LTL :

```
[] ~(PutingItem("P") /\ GetingItem("C"))
```

- Description :

  En démarrant la vérification à partir de l'état initial *initial1* (voir la figure 5.49), cette propriété vérifie que l'exclusion mutuelle est satisfaite. En d'autres termes, cela signifie qu'il n'est pas possible de retrouver à la fois le Producteur et le Consommateur dans leurs sections critiques respectives.

Propriété 6 :

- Formule LTL :

```
[] <> ( PutingItem("P") ->  ConsumerSleeping("C"))
```

- Description :

  En démarrant la vérification à partir de l'état initial *initial1* (voir la figure 5.49), cette propriété est infiniment souvent vraie : lorsque le Producteur est en train de transmettre des informations à l'objet *Buffer*, le Consommateur est dans son état de veille (*ConsumerWaiting*). Cette propriété a donc pour effet de vérifier le bon comportement des objets lorsqu'ils transmettent des informations à un objet *Buffer*.

Propriété 7 :

- Formule LTL :

  ```
  [] <> ( GetingItem("C") ->  ProducerSleeping("P"))
  ```

- Description :

  En démarrant la vérification à partir de l'état initial *initial1* (voir la figure 5.49), cette propriété est infiniment souvent vraie : lorsque le Consommateur est en train de lire des informations de l'objet *Buffer*, le Producteur est dans son état de veille (*ProducerWaiting*). Cette propriété a donc pour effet de vérifier le bon comportement des objets lorsque la lecture des informations d'un objet *Buffer* est en cours.

La figure 5.48 présente une partie du module *COMMUNICATION-PREDICATES* dans lequel sont définis tous ces prédicats. Les prédicats montrés dans la figure sont limités à ceux relatifs à l'objet *P* de la classe *Producer*.

```
mod COMMUNICATION-PREDICATES is
  protecting COMMUNICATION .  including SATISFACTION .
  subsort Configuration < State .
  var Cf : Configuration .
  *** Operators
  ops PutingItem ProducerSleeping : Poid -> Prop .
  ops GetingItem ConsumerSleeping : Coid -> Prop .
  *** Producer's Predicates
  eq  < "P" : Producer | StateP : Producing, ItemP : 5 >
    < "B" : Buffer | StateB : Empty, ItemB : NoItemB,
      Semaphore : 1 > Cf |= PutingItem ("P") = true . *** [1]
  eq < "P" : Producer | StateP : ProducerWaiting, ItemP :
      NoItemP > Cf |= ProducerSleeping("P") = true .  *** [2]
  ...
endm
```

FIG. 5.48 – Le module *COMMUNICATION-PREDICATES*

Un prédicat est défini pour chaque état de chaque classe. Comme la classe *Producer* avait deux états possibles, *Producing* et *ProducerWaiting* (voir le diagramme d'états-transitions de la figure 5.36.(a)), deux prédicats sont définis. Les lignes 1 et 2 représentent les prédicats pour chacun de ces deux états.

Vérification des propriétés

Un peu plus haut, lorsque les sept propriétés ont été énoncées, il a été question d'un état initial *initial1* par lequel doit débuter la vérification de chacune d'entre elles. En plus d'introduire cette configuration initiale, le module *COMMUNICATION-CHECK*, montré à la figure 5.49, établit le lien final entre le système étudié et le vérificateur de modèles de Maude.

```
mod COMMUNICATION-CHECK is
 including COMMUNICATION-PREDICATES .
 including MODEL-CHECKER LTL-SIMPLIFIER .
 op initial1 : -> Configuration .
 eq initial1 = ComingMsg(Produce( EmptyParameterList), "P")
               ComingMsg( Get(5), "B")
  < "P" : Producer | StateP : ProducerWaiting, ItemP :
                     NoItemP >
  < "C" : Consumer | StateC : ConsumerWaiting, ItemC :
                     NoItemC >
  < "B" : Buffer | StateB : Empty, ItemB : NoItemB,
                     Semaphore : 1 > .
endm
```

FIG. 5.49 – Le module *COMMUNICATION-CHECK*

La figure 5.50 montre quelques-uns des appels à la fonction *modelChekc* de Maude utilisés pour la vérification des sept propriétés énoncées précédemment.

```
red modelCheck(initial1, [] PutingItem("P")) .
...
red modelCheck(initial1, [] <> ( GetingItem("C") ->
                ProducerSleeping("P")))
```

FIG. 5.50 – Quelques appels à la fonction *modelCheck*.

Les résultats obtenus sont montrés en partie par la capture d'écran de la figure 5.51. Les sept résultats sont également donnés dans le tableau 5.5. On remarque alors que sur l'ensemble des propriétés, cinq d'entre elles ont été évaluées à Vrai (*True*), tandis que deux autres montrent un résultat de Contre-exemple signifiant qu'elles n'ont pas été vérifiées. Ce résultat de contre-exemple, dont un aperçu est donné à la figure 5.51, montre le chemin d'exécution

TAB. 5.5 – Résultats de l'évaluation des sept propriétés du problème du Producteur – Consommateur

Propriété	Résultat
1	Contre-exemple
2	Vrai
3	Contre-exemple
4	Vrai
5	Vrai
6	Vrai
7	Vrai

que le vérificateur a pris avant de rencontrer un état d'erreur. Ce résultat pourra être fort utile lorsque viendra le temps d'apporter les correctifs nécessaires aux modèles UML.

Analyse des résultats et Conclusion

Dans ce qui suit, les résultats de la vérification de modèles effectuées dans le cadre du problème du Producteur – Consommateur sont analysés et les conclusions pertinentes sont tirées.

Résultats du comportement individuel. Deux propriétés ont été définies ayant pour objectif la vérification du comportement de *P* (Propriétés 1 et 2). Selon les résultats obtenus, l'objet *P* se comporte adéquatement. En effet, la Propriété 1 visait à vérifier si *P* était continuellement dans sa section critique. Comme le résultat donné par Maude est un contre-exemple, cela signifie que l'objet *P* de la classe *Producer* n'est pas continuellement dans son état *Producing*, ce qui constitue un comportement parfaitement acceptable.

En ce qui concerne la Propriété 2, toujours en rapport avec *P*, elle a été évaluée à vrai. Cette propriété exprimait le fait que le système ne permettait pas à l'objet *P* d'être toujours dans son état *ProducerWaiting* (en veille). Ceci constitue également un comportement accep-

table au sein du système étudié, prévenant ainsi un blocage au niveau du Producteur.

Les propriétés définies en vue de vérifier le comportement individuel de l'objet *C* de la classe *Consumer* sont analogues à celles du Producteur. La Propriété 3 tentait de vérifier si le système permettait à l'objet *C* d'être perpétuellement dans sa section critique (dans son état *Consuming*). De façon analogue à la Propriété 1, Maude a évalué cette propriété comme étant fausse en fournissant un contre-exemple. Ainsi, le système ne permet pas au Consommateur d'être éternellement en mode lecture des informations de la ressource partagée.

La Propriété 4, analogue à la Propriété 2, exprimait le fait que le système ne permet pas au Consommateur d'être toujours en attente. Comme le résultat est positif (*True*), le système ne permet pas cette situation, et le Consommateur a un comportement correct.

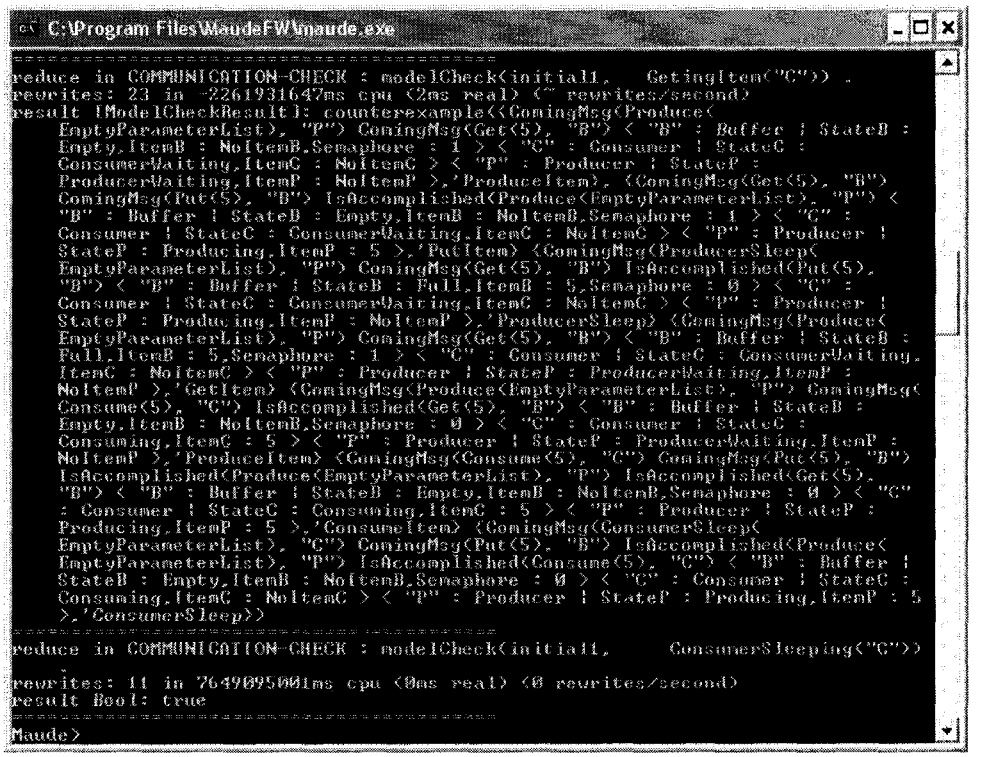


FIG. 5.51 – Partie des résultats des appels à la fonction *modelCheck*

Résultats du comportement collectif. Les trois dernières propriétés étaient quant à elles définies pour vérifier le comportement collectif des objets *P*, *C* et *B*. Toutes ces propriétés ont obtenu un résultat positif (*True*) : elles ont alors toutes été vérifiées. De façon plus spécifique, la Propriété 5 vérifiait si le système permettait aux deux objets actifs concurrents, le Producteur et le Consommateur, d'être tous les deux dans leur section critique en même temps. Ceci signifierait que le Producteur pourrait être en train d'écrire des informations au niveau de la ressource partagée au moment même où le Consommateur tenterait d'y lire des informations. Ceci aurait pour effet de créer des inconsistences au niveau des données. Heureusement, le système ne permet par ce type de situation.

Quant à elles, les Propriétés 6 et 7 expriment des contreparties :

- La Propriété 6 exprime le fait que lorsque le Producteur est en train d'écrire des informations dans la ressource partagée, le Consommateur, lui, est en mode d'attente ;
- La Propriété 7 exprime le fait que lorsque le Consommateur est en train de lire des informations de la ressource partagée, le Producteur, lui, est en mode d'attente.

Ces deux propriétés ont toutes deux été évaluées à Vrai (*True*). Cela signifie alors que le comportement collectif de ces deux objets est correct : lorsqu'un des deux objets actifs concurrents est dans sa section critique, l'autre est en veille.

Conclusion

En Résumé

Dans ce mémoire, les techniques de spécification formelle et de vérification de modèles ont été revues afin d'en présenter les notions et concepts importants. Ces deux techniques ont fait l'objet d'un chapitre chacun, respectivement les chapitres 1 et 2, sous la forme d'une revue de la littérature, tout en résumant les principales techniques présentement reconnues.

L'environnement Maude a également fait l'objet d'une étude approfondie au chapitre 3, notamment sur ses fondements mathématiques et son langage. Maude est un langage de spécification et de programmation basé sur deux types de logique : la logique des équations ensemblistes et la logique de réécriture. Cette dernière a été tout particulièrement développée en vue de spécifier plus aisément les systèmes concurrentiels, qui sont aujourd'hui un type de système très répandu.

Ce mémoire avait pour objectif de développer un cadre formel pour la translation et la vérification de diagrammes UML à l'aide de l'environnement Maude. Ce processus a été décrit au chapitre 4. Ce dernier a par la suite été appliqué à trois études de cas différentes. Le premier était un système OO classique de gestion d'un système d'ascenseur. Le second SOO, un système de guichet automatique, montrait des éléments de concurrence interne au niveau d'une de ses classes. Enfin, la dernière étude de cas portait sur un SOOC, un système de Producteur – Consommateur, un exemple très connu dans le domaine de l'informatique.

Approche proposée

Les objectifs poursuivis tout au long de ce mémoire étaient de développer un cadre formel par lequel il serait possible de translater des diagrammes UML vers une notation formelle du langage Maude. Par la suite, cette description formelle Maude sera validée à l'aide de deux des outils de l'environnement Maude : les simulations et la vérification de modèles.

Le processus développé se divise en quatre étapes majeures :

1. La description d'un système OO à l'aide de diagrammes UML. Les diagrammes UML considérés par l'approche proposée sont les suivants :
   - Diagrammes de classes UML ;
   - Diagrammes d'états-transitions UML ;
   - Diagrammes de communication UML.

   La prise en charge de ces trois types de diagrammes permet de considérer à la fois les aspects statiques et dynamiques des systèmes OO.
   - Les aspects statiques d'un système OO sont en réalité la structure en classes décrits à l'aide des diagrammes de classes UML ;
   - Les aspects dynamiques réfèrent au comportement des objets du système. Ils sont de deux natures :
     - Le comportement individuel des objets réfère au comportement qu'adopte un objet donné à la réception d'un message donné. Ce comportement interne est décrit à l'aide de diagrammes d'états-transitions UML ;
     - Le comportement collectif des objects réfère au comportement qu'adopte un groupe d'objets en vue de l'accomplissement d'une tâche qui leur est dévolue. Ce comportement est décrit en termes d'interactions dynamiques par des diagrammes de communication UML.
2. La validation inter-diagramme entre les divers diagrammes UML développés pour décrire le modèle du système. Cette étape de vérification consiste principalement à s'assurer que tous les bons éléments sont présents dans chacun des diagrammes. Par exemple, un message envoyé au sein d'un diagramme de communication doit figurer dans le dia-

gramme de classes. De plus, ce que l'objet récepteur fait lorsqu'il reçoit ce message doit aussi être décrit dans le diagramme d'états-transitions correspondant à sa classe ;

3. Génération d'une description formelle Maude. Cette étape consiste à dériver systématiquement une description formelle basée sur le langage Maude à partir des trois types de diagrammes UML mentionnés plus haut ;
4. Vérification de la description formelle du système à l'aide du vérificateur intégré de l'environnement Maude. Cette étape se fait à l'aide du vérificateur intégré de l'environnement Maude et des propriétés comportementales du système énoncées à l'aide de la logique LTL.

Ce processus en quatre étapes a été développé à l'origine seulement pour les SOO traditionnels. Il a cependant été amélioré et étendu pour prendre en considération les systèmes concurrentiels. La concurrence interne des objets (avec l'utilisation de régions orthogonales concurrentes dans leur diagramme d'états-transitions) et la concurrence externe (avec les objects actifs) sont tous les deux pris en charge par l'approche. Les SOOC sont en effet aujourd'hui très répandus : le paradigme de programmation concurrentielle et le paradigme de programmation orienté-objet sont en effet compatibles et, combinés, offrent des possibilités très intéressantes. Il était donc important de les inclure dans l'approche développée.

Évaluation de l'approche

Application. L'approche proposée a été validée à l'aide de trois études de cas différents. Ces trois études de cas étaient, dans l'ordre :

- Un **système de gestion d'un ascenseur** : Ce système est un SOO classique ;
- Un **système de guichet automatique** : Ce système a la particularité de montrer des aspects de concurrence interne. En effet, un des diagrammes d'états-transitions de cet exemple montre des régions orthogonales concurrentes dans un état composite ;
- Un **système de Producteur – Consommateur** : Ce système est un SOOC pur. Les objets de classes Producteur et Consommateur sont des objets actifs qui tentent d'accéder

à une ressource commune partagée.

Comparaison. Dans ce qui suit, la translation développée dans ce mémoire est comparée à certaines autres approches proposées dans la littérature.

Comparativement à quelques-unes des techniques étudiées au chapitre 1, le processus de translation adopté ici intègre à la fois les aspects statiques d'un SOO (diagramme de classes UML), le comportement individuel de ses objets (diagramme d'états-transitions UML), ainsi que leur comportement collectif en termes d'interactions dynamiques (diagramme de communication UML). Les autres approches décrites ne considèrent, quant à elles, qu'un ou deux de ces aspects.

Seule la combinaison de trois approches différentes utilisant le langage *Object-Z* pouvait en arriver à une représentation relativement complète d'un SOO. Le lecteur est référé au chapitre 1 pour de plus amples détails à ce sujet.

Tel que mentionné au chapitre 1, plusieurs approches proposent des techniques basées sur des modèles développés à partir de l'environnement CASE *Rational Rose*. En effet, *Rational Rose* est un des outils CASE les plus populaires. Il est donc normal que plusieurs approches l'utilisent. C'est le cas notamment des approches proposées par Snook et al. [82] et Laleau et Mammar [50, 49]. Cependant, ces approches souffrent de certaines insuffisances d'extensibilité. Ces manquements sont essentiellement liées à la difficulté de manipuler les modèles UML avec le langage script de *Rational Rose*.

D'autre part, il avait été mentionné au chapitre 3 que Maude est un langage très expressif et très puissant. Il est en effet plus puissant et plus expressif que des langages tels *Z*, *Object-Z* et *B*. Ces langages ont des structures de notations rigides, tandis que Maude permet la création d'opérateurs et de notations personnalisées.

Selon D'Inverno [25], le langage *Z* est inapproprié pour la modélisation de systèmes complexes. La description des systèmes complexes à l'aide d'une spécification du langage *Object-Z* est pratiquement toujours complétée à l'aide d'autres notations permettant de capturer les autres points de vue du système. De plus, Duran [28] mentionne que la logique sous-jacente au langage *Object-Z* n'est pas assez expressive. Enfin, la notation *B* n'est pas

orientée-objet, limitant du fait même son utilisation avec les SOO. En effet, la spécification abstraite engendrée à partir d'une modélisation objet est très différente de celle qui aurait été écrite directement. Il est donc nécessaire d'ajouter les éléments manquants manuellement, établir les preuves et poursuivre le processus par l'écriture des raffinements, comme le mentionnent Hammad et Tatihouet [36].

Limites. La méthode proposée dans ce mémoire présente quelques limites. Tout d'abord, le processus de validation inter-diagramme (Étape 2) est fait manuellement pour le moment. Cette vérification pourra très certainement un jour être automatisée.

De plus, la génération du code source Maude lors de l'application de la translation de diagramme UML vers une notation formelle Maude (Étape 3) se veut automatique. L'apport de ce mémoire consiste en développer la notation adéquate pour représenter formellement un SOO. Une équipe d'ingénieur est actuellement à développer un outil pour la translation automatique d'UML vers Maude. L'outil est fonctionnel, sans toutefois être terminé.

Le nombre de diagrammes UML considérés est limité à trois. Il existe cependant de nombreux autres diagrammes UML, tout particulièrement avec la version 2.0 d'UML. De plus, ces diagrammes sont des structures parfois très riches exprimant un grand nombre de caractéristiques d'un système. Un simple exemple est le diagramme d'états-transitions. Ce type de diagramme est très riche par ses notations d'états et de transitions, les instructions d'entrée et de sortie, etc. Seuls les éléments de base des trois diagrammes sont considérés :

- Pour les diagrammes de classes UML, l'agrégation et l'héritage sont pris en charge ;
- Pour les diagrammes d'états-transitions UML, les états, les transitions et les gardes peuvent être translatés ;
- Pour les diagrammes de communication, les messages synchrones et asynchrones sont pris en charge, ainsi que les points de synchronisation.

Enfin, l'approche proposée ne considère qu'un seul diagramme de communication à la fois. Un système d'envergure comprendra bien évidemment plusieurs de ces diagrammes. Il suffira alors d'appliquer le même processus à plusieurs reprises, un diagramme de communication à la fois, pour valider un système en entier. La structure statique du système sera déjà

décrite à ce moment, et ce bout de code source Maude n'aura pas à être repris à chaque fois, dans la mesure où aucun correctif n'est à apporter.

Travaux Futurs

La notation appropriée pour la translation de SOO traditionnels ou concurrentiels a été introduite dans ce mémoire. Cette notation a été testée sur trois études de cas simples couvrant ces types de systèmes. Quelques éléments sont toutefois à compléter et améliorer.

Tout d'abord, il serait intéressant de développer un outil pour automatiser l'étape 2 du processus proposé, soit la validation inter-diagrammes. Cet outil permettrait de passer beaucoup plus rapidement à la génération d'une description formelle Maude et accélérerait grandement le processus de validation.

Ensuite, et c'est probablement un enjeu majeur, l'outil permettant de translater automatiquement les diagrammes UML en une notation formelle Maude devra être affiné et amélioré.

De plus, il serait fort intéressant de vérifier quels autres diagrammes UML pourrait être intégrés à la méthode. Plusieurs de ces diagrammes capturent des aspects intéressants des SOO. Intégrer ces aspects à une description formelle aura pour effet d'améliorer la validité des modèles développés.

Qui plus est, la méthode n'offre pas de support pour plusieurs des éléments importants des diagrammes UML considérés. L'intégration de ces éléments à la description formelle Maude générée serait un atout.

Enfin, la promotion d'une telle approche auprès des analystes est également un enjeu majeur. Plusieurs analystes considèrent toujours les méthodes formelles comme une étape supplémentaire dont les efforts requis ne sont pas justifiés. Bien au contraire ! Les bénéfices sont bien présents. La vérification de modèles s'établit petit-à-petit comme le nouveau standard d'assurance qualité de l'industrie. Rendre une telle technique automatique et directement applicable est le grand objectif de plusieurs travaux de recherche.

Références bibliographiques

[1] Akehurst, D., Derrick, J. and Borten, E. A Framework for UML Consistency. In *Proceedings of the IEEE Workshop on Consistency Problems in UML-based Software Development*, Department of Software Engineering and Computer Science of Blekinge Institute of Technology, pages 30–45, 2002.

[2] Araujo, J. and Moreira, A. Specyfying the Behavior of UML Collaborations Using Object-Z. Departamento de Infomatica, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal, 2000.

[3] ArgoUML Tigris Organisation. *ArgoUML User Manual*, 2002.

[4] Astesiano, E. UML as Heterogeneous Multiview Notation. Strategie for a Formal Foundation. In L. Andrade, A. Moreira, A. deshpande, and S. Kent, editors, *Proceedings of the Conference on Object Oriented programming, Systems, Languages and Applications (OOPSLA'98) - Workshop on Formalizing UML. Why ? How ?*, Canada, 1998.

[5] Barnett, M., DeLine, R., Fahndrich, M., Rustan M. Leino, K. and Schulte, W. Verification of Object-Oriented Programs with Invariants. In *Formal Techniques for Java-like Programs*, Available as Technical Report 408, Department of Computer Science, ETH, July 2003.

[6] Biere, A., Cimatti, A., Clarke, E. M., trichman, O. and Zhu, Y. Bounded Model Checking. *Advances in Computers*, 58, 2003.

[7] Bowen, J. *Formal Specification and Documentation Using Z : A Case Study Approach.* 2003.

[8] Börger, E., Cavarra, A. and Riccobene, E. Modeling the Dynamics of UML State Machines. In *ASM '00 : Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, Springer-Verlag, pages 223–241, 2000.

[9] Börger, E., Cavarra, A. and Riccobene, E. The Meaning of Concurrent States in UML State Diagrams. In *SAC 2003*, Springer-Verlag, pages 223–241, 2003.

[10] Canals, A., Cassaing, Y., Jammes, A., Pomiès, L. and Roblet, E. How You Could Use NEPTUNE in the Modelling Process. In *Journal of Object Technology*, 2(1) :69–83, 2003.

[11] Cansell, D. and Méry, D. Tutorial on the Event-based B Method. In *FORTE 2006 : Proceedings of the 26th International Conference on Formal Methods for Networked and Distributed Systems*, Paris, France, 2006.

[12] Chan, W., Anderson, R. J., Beame, P., Burns, S., Modugno, F., Notkin, D. and Reese, J. D. Model Checking Large Software Specifications. *IEEE Transactions on Software Engineering*, 24(7) :498–520, July 1998.

[13] Chan, W., Anderson, R. J., Beame, P. and Notkin, D. Improving Efficiency of Symbolic Model Checking for State-based System Requirements. In Michal Young, editor, *ISSTA 98 : Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 102–112, Clearwater Beach, Florida, USA, March 1998. ACM.

[14] Chechik, M. A. *Automatic Analysis of Consistency Between Requirements and Designs.* PhD thesis, University of Maryland at College Park, 1996.

[15] Chechik, M. A., Devereux, B. and Gurfinkel, A. Model-checking Infinite State-space Systems with Fine-grained Abstractions Using SPIN. In *Lecture Notes in Computer Science*, 2057 :16+, 2001.

[16] Chechik, M. and Gannon, J. D. Automatic Analysis of Consistency Between Requirements and Designs. In *Software Engineering*, 27(7) :651–672, 2001.

[17] Clarke, E. M., Emerson, E. A. and Sistla, A. P. Automatic Verification of Finite State Concurrent System Using Temporal Logic Specifications : A Practical Approach. In *POPL '83 : Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 117–126, New York, NY, USA, 1983. ACM Press.

[18] Clarke, E.M., Grumberg, O. and Long, D. E. Verification Tools for Finite-state Concurrent Systems. In *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, pages 124–175, London, UK, 1994. Springer-Verlag.

[19] Clarke, E. M., Wing, J. M., Alur, R., Cleaveland, R., Dill, D., Emerson, A., Garland, S., German, S., Guttag, J., Hall, A., Henzinger, T., Holzmann, G., Jones, C., Kurshan, R., Leveson, N., McMillan, K., Moore, J., Peled, D., Pnueli, A., Rushby, J., Shankar, N., Sifakis, J., Sistla, P., Steffen, B., Wolper, P., Woodcock, J. and Zave, P. Formal Methods : State of the Art and Future Directions. In *ACM Computing Surveys*, 28(4) :626–643, 1996.

[20] Clarke, E. M. Model Checking Overview. Presentation Slides, 2003.

[21] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. and Quesada, J. F. Maude : Specification and Programming in Rewriting Logic. In *Theoretical Computer Science*, 2001.

[22] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Mesenguer, J. and Talcott, C. *Maude Manual (Version 2.1.1)*, April 2005.

[23] Das, B., Sarkar, D. and Chattopadhyay, S. Model Checking on State Transition Diagram. In *ASP-DAC '04 : Proceedings of the 2004 Conference on Asia South Pacific Design Automation*, pages 412–417, Piscataway, NJ, USA, 2004. IEEE Press.

[24] del-Mar Gallardo, M., Merino, P. and Pimentel, E. Debugging UML Designs With Model Checking. In *Journal of Object Technology*, Vol. 1(No. 2) :pages 101–117, 2002.

[25] d'Inverno, M., Kinny, D. and Luck, M. Interaction Protocols in Agentis, 1998. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS98)*, pages 261–268, 1998.

[26] Dong, J. S. Formal Specification and Design Techniques. Lecture Notes, 2000.

[27] Duke, R., Rose, G. and Smith, G. Object-Z : A Specification Language Advocated for the Description of Standards. In *Comput. Stand. Interfaces*, 17(5-6) :511–533, Elsevier Science Publishers B. V., 1995.

[28] Duran, F. and Vallecillo, A. Writing ODP Enterprise Specifications in Maude, 2001. Technical Report ITI-2001-8, Departmento de Lenguajes y Ciencias de la Computacion, University of Malaga, Apr. 2001.

[29] Eker, S., Meseguer, J. and Sridharanarayanan, A. The Maude LTL Model Checker. F. Gaducci and U. Montanari, editors, *Proc. of the 4th International Workshop on Rewriting Logic and its Applications (WRLA 2002)*, 2002.

[30] Favre, L. Foundations for MDA-based Forward Engineering. In *Journal of Object Technology*, Vol. 4(No. 1) :pages 129–153, 2005.

[31] Funes, A. and George, C. Formal Foundations in RSL for UML Class Diagrams. Technical Report 253. UNU/IIST, May 2002.

[32] Gagnon, P., Mokhati, F. and Badri, M. Applying Model Checking to Concurrent UML Models. In *Journal of Object Technology*, 7(1) : January – February 2008 (To be published).

[33] García Roselló, E., Ayude, J., García-Schofield, J. B. and Pérez Cota, M. Design Principles for Highly Reusable Concurrent Object-Oriented Systems. In *Journal of Object Technology*, 1(1) :107–123, 2002.

[34] Giannakopoulou, D. and Magee, J. Fluent Model Checking for Event-based Systems. In *ESEC/FSE-11 : Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 257–266, New York, NY, USA, 2003. ACM Press.

[35] Gomaa, H. *Designing Concurrent, Distributed, and Real-Time Applications with UML.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[36] Hammad, A. and Tatibouet, B. Spécifications formelles et semi-formelles : l'exemple du robot type. In *Fifth International Symposium on Programming and Systems (ISPS'2001)*, pages 229–240, Algiers, Algeria, May 2001.

[37] Havelund, K. *Java PathFinder User Guide*, August 1999.

[38] Havelund, K. and Skakkebaek, J. U. Applying Model Checking in Java Verification. In *SPIN*, pages 216–231, 1999.

[39] Havelund, K. and Pressburger, T. Model Checking JAVA Programs Using JAVA Pathfinder. In *International Journal on Software Tools for Technology Transfer*, 2(4) :366–381, 2000.

[40] Holzmann, G. J. Software Analysis and Model Checking. In *CAV '02 : Proceedings of the 14th International Conference on Computer Aided Verification*, pages 1–16, London, UK, 2002. Springer-Verlag.

[41] Huizing, K. and Kuiper, R. Verification of Object Oriented Programs Using Class Invariants. In *FASE '00 : Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering*, pages 208–221, London, UK, 2000. Springer-Verlag.

[42] Jansamak, S. and Surarerks, A. Formalization of UML Statechart Models Using Concurrent Regular Expressions. In *CRPIT '04 : Proceedings of the 27th Conference on Australasian Computer Science*, pages 83–88, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[43] Johnston, W. A Type Checker for Object-Z. Technical Report 96–24, SVRC, University of Queensland, September 1996.

[44] Kaveh, N. and Emmerich, W. Deadlock Detection in Distribution Object Systems. In *ESEC/FSE-9 : Proceedings of the 8th European Software Engineering Conference Held Jointly With 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 44–51, New York, NY, USA, 2001. ACM Press.

[45] Kholkar, D., Murali Krishna, G., Shrotri, U. and Venkatesh, R. Visual Specification and Analysis of Use Cases. In *SoftVis '05 : Proceedings of the 2005 ACM Symposium on Software Visualization*, pages 77–85, New York, NY, USA, 2005. ACM Press.

[46] King, P. *Printing Z and Object-Z LaTeXDocuments*. University of Queensland, Australia, 4072, 1990.

[47] Kotb, Y. and Katayama, T. Consistency Checking of UML Model Diagrams Using the XML Semantics Approach. In *WWW '05 : Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, pages 982–983, New York, NY, USA, 2005. ACM Press.

[48] Kupferman, O. and Vardi, M. Y.. An Automata-Theoretic Approach to Modular Model Checking. In *ACMTOPLAS : ACM Transactions on Programming Languages and Systems*, 22, 2000.

[49] Laleau, R. and Mammar, A. A Generic Process to Refine a B Specification Into a Relational Database Implementation. In *ZB '00 : Proceedings of the First International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 22–41, London, UK, 2000. Springer-Verlag.

[50] Laleau, R. and Mammar, A. An Overview of a Method and its Support Tool for Generating B Specifications from UML Notations. In *ASE '00 : Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, page 269, Washington, DC, USA, 2000. IEEE Computer Society.

[51] Lam, V. S. W. and Padget, J. A. Symbolic Model Checking of UML Statechart Diagrams with an Integrated Approach. In *ECBS*, pages 337–347. IEEE Computer Society, 2004.

[52] Lam, V. S. W. and Padget, J. A. An Integrated Environment for Communicating UML Statechart Diagrams. IEEE Computer Society, 2005.

[53] Ledang, H., Souquières, J. and Charlesé, S. ArgoUML+B : un outil de transformation systématique de spécifications UML en B. Dans *Actes de la conférence AFADL'03*, INRIA, pages 15–17, Rennes, France, Janvier 2003.

[54] Levin, V. Xie, F. and Browne, J. C. Integrating Model Checking Into Object-Oriented Software Development Processes. Technical Report TR-01-03, UTCS, 2001.

[55] MacColl, I. and Carrington, D. A. Specifying Interactive Systems in Object-Z and CSP. In *IFM*, pages 335–352, 1999.

[56] Malgouyres, H., Jean-Pierre, S. W. and Motet, G. Identification de Règles de Cohérence d'UML 2.0. Journée SEE "Systèmes Informatiques de Confiance" sur le thème "Vérification de la cohérence de modèles UML", Paris, France, Mars 2005.

[57] Martin, R. C. UML Tutorial : Part 1 – Class Diagrams. In *Engineering Notebook Column*, 1997.

[58] Martin, R. C. UML Tutorial : Collaboration Diagrams. In *Engineering Notebook Column*, 1997.

[59] McCombs, T. Maude 2.0 Primer (Version 1.0). August 2003.

[60] Merz, S. Model Checking : A Tutorial Overview. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, *MOVEP*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer, 2000.

[61] Meseguer, J. Rewriting as a Unified Model of Concurrency. In *SIGPLAN OOPS Mess.*, 2(2) :86–88, 1991.

[62] Meseguer, J. A Logical Theory of Concurrent Objects and its Realization in the Maude Language. In *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, Cambrige, MA, USA, pages 314–390, 1993.

[63] Meseguer, J. Software Specification and Verification in Rewriting Logic. Germany, August 2002.

[64] Meseguer, J. Software Specification and Verification in Rewriting Logic. 2003.

[65] Mokhati, F., Badri, M. and Gagnon, P. Translating UML Diagrams into Maude Formal Specification : A Systematic Approach. In *SEKE'06 : Proceedings of the 18th International Conference of Software Engineering and Knowledge Engineering*, San Francisco, 2006.

[66] Moreira, A., Bruel, J. M., Lilius, J. and Robert, B. Defining Precise Semantics for UML. In *ECOOP'2000 Workshop Reader*, number 1964. Springer-Verlag, November 2000.

[67] Müller-Olm, M., Schmidt, D. and Steffen, B. Model Checking : A Tutorial Introduction. In *Lecture Notes in Computer Science*, 1694 :330–354, 1999.

[68] Muller, P. A. and Gaertner, N. *Modélisation Objet avec UML.* Deuxième édition, Paris, 2000.

[69] Muskens, J., Somers, L., Lange, C., Chaudron, M. and Dortmans, E. An Empirical Investigation in Quantifying Inconsistency and Incompletness in UML Designs. In *Proceedings of the IEEE Workshop on Consistency Problems in UML-Based Software Development II*, pages 26–34. IEEE and Department of Software Engineering and Computer Science of Blekinge Institute of Technology, 2003.

[70] Naixiao, Z., Meng, S. and Aichernig, B. K. The Formal Foundations in RSL for UML Statechart Diagrams. Technical Report 299. UNU/IIST, 2004.

[71] Object Modeling Group. *Unified Modeling Language Specification, Version 1.4.* September 2001.

[72] Object Modeling Group. *Unified Modeling Language Specification, Version 2.0.* July 2005.

[73] Ölveczky, P. C. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic.* PhD thesis, University of Bergen, 2000.

[74] Paige, R. F., Kaminskaya, L., Ostroff, J. S. and Lancaric, J. BON-CASE : An Extensible CASE Tool for Formal Specification and Reasoning. In *Journal of Object Technology*, 1(3) :77–96, 2002.

[75] Paige, R. F. and Brooke, P. J. Integrating BON and Object-Z. In *Journal of Object Technology*, 3(3) :121–141, 2004.

[76] Reggio, G. and Wieringa, R. Thirty One Problems in the Semantics of UML 1.3 Dynamics. In *Conference on Object Oriented programming, Systems, Languages and Applications (OOPSLA'99) - Workshop "Rigorous Modeling an Analysis of the UML Challenges and Limitations"*, 1999.

[77] Rumbaugh, J. and Jacobson G. Booch, I. In *The Unified Modeling Language User Guide.* Addition-Wesley, Object Technology Series, 1998.

[78] Schäfer, T., Knapp, A. and Merz, S. Model Checking UML State Machines and Collaborations. In *Electronic Notes in Theoretical Computer Science*, 55(3) :13 pages, 2001.

[79] Schmidt, J. W. and Matthes, F. State diagrams. 1999.

[80] Shrotri, U., Bhaduri, P. and Venkatesh, R. Model Checking Visual Specification of Requirements. In *SEFM*, pages 202–209. IEEE Computer Society, 2003.

[81] Smith, G. and Duke, R. Specifying Concurrent Systems Using Object-Z. In *Proceedings 15th Australian Computer Science Conference (ACSC-15)*, pages 859–871, January 1992. 148",

[82] Snook, C. and Butler, M. U2B - A Tool for Translating UML-B Models Into B. In *UML-B Specification for Proven Embedded Systems Design*, Springer, 2004.

[83] Taïani, F., Paludetto, M. and Cros, T. Model Checking and Object Orientation : A Tool Overview. 2000.

[84] van Lamsweerde, A. Formal Specification : A roadmap. In *ICSE - Future of SE Track*, pages 147–159, 2000.

[85] Wahl, T. (Yet Another) Introduction to Model Checking. Presentation Slides, Spring 2003.

[86] Warmer, J. B. and Kleppe, A. G.. *The Objec Constraint Language – Precise Modeling with UML.* Addison-Wesley, 1999.

[87] Wegner, P. Concepts and Paradigms of Object-Oriented Programming. In *SIGPLAN OOPS Mess.*, 1(1) :7–87, 1990.